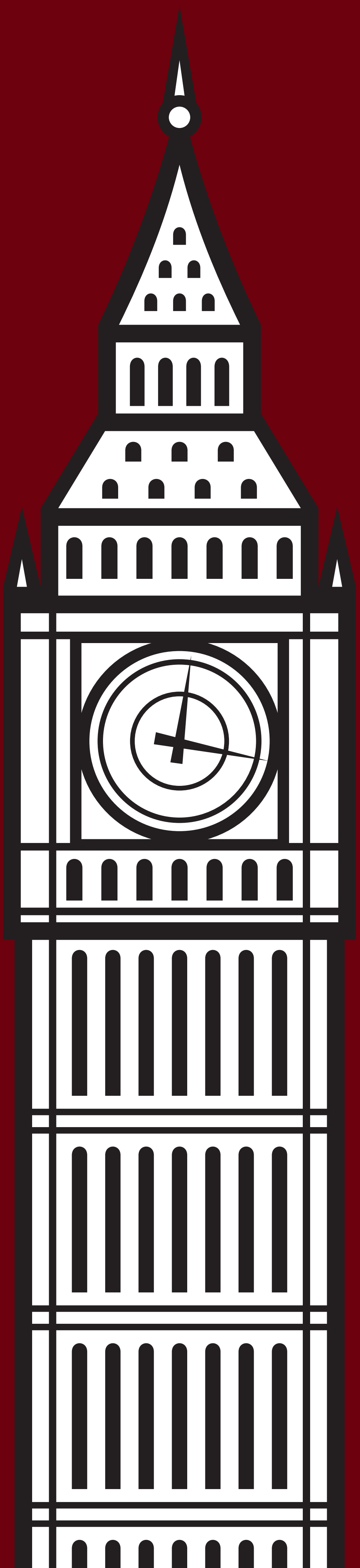


XML LONDON 2014

CONFERENCE PROCEEDINGS

UNIVERSITY COLLEGE LONDON,
LONDON, UNITED KINGDOM

JUNE 7-8, 2014

XML London 2014 – Conference Proceedings
Published by XML London
Copyright © 2014 Charles Foster

ISBN 978-0-9926471-1-7

Table of Contents

Are you stuck fixing

YESTERDAY

Or are you solving for

TOMORROW?

The world's largest banks use MarkLogic to get a 360-degree view of the enterprise, reduce risk and operational costs, and provide better customer service. What are you waiting for?

Hello, we're your Enterprise NoSQL database solution.

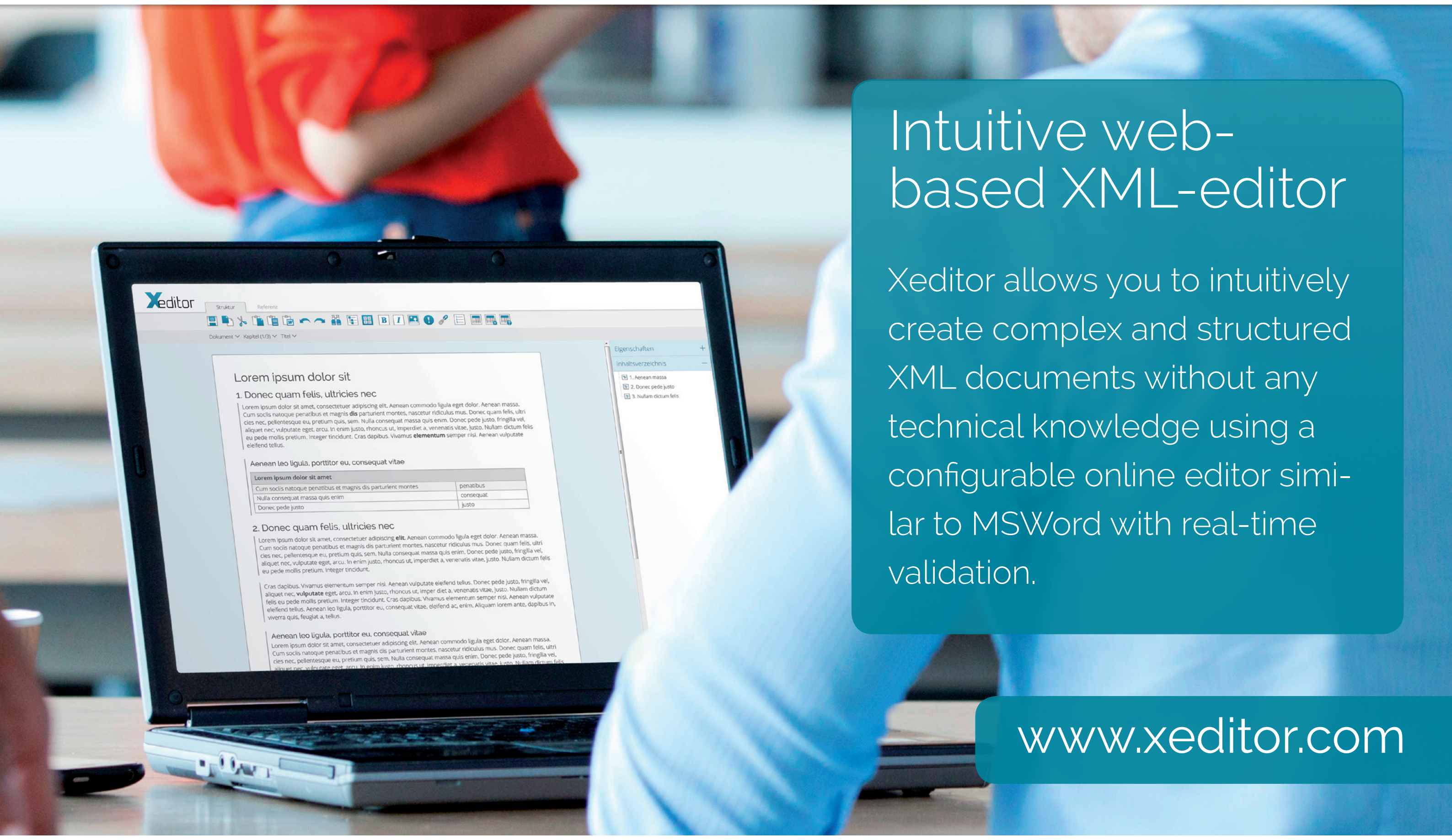
MarkLogic®

Say hello to the new generation.

POWERFUL & SECURE | AGILE & FLEXIBLE | ENTERPRISE-READY | TRUSTED

www.marklogic.com · sales@marklogic.com

Xeditor
professional authoring

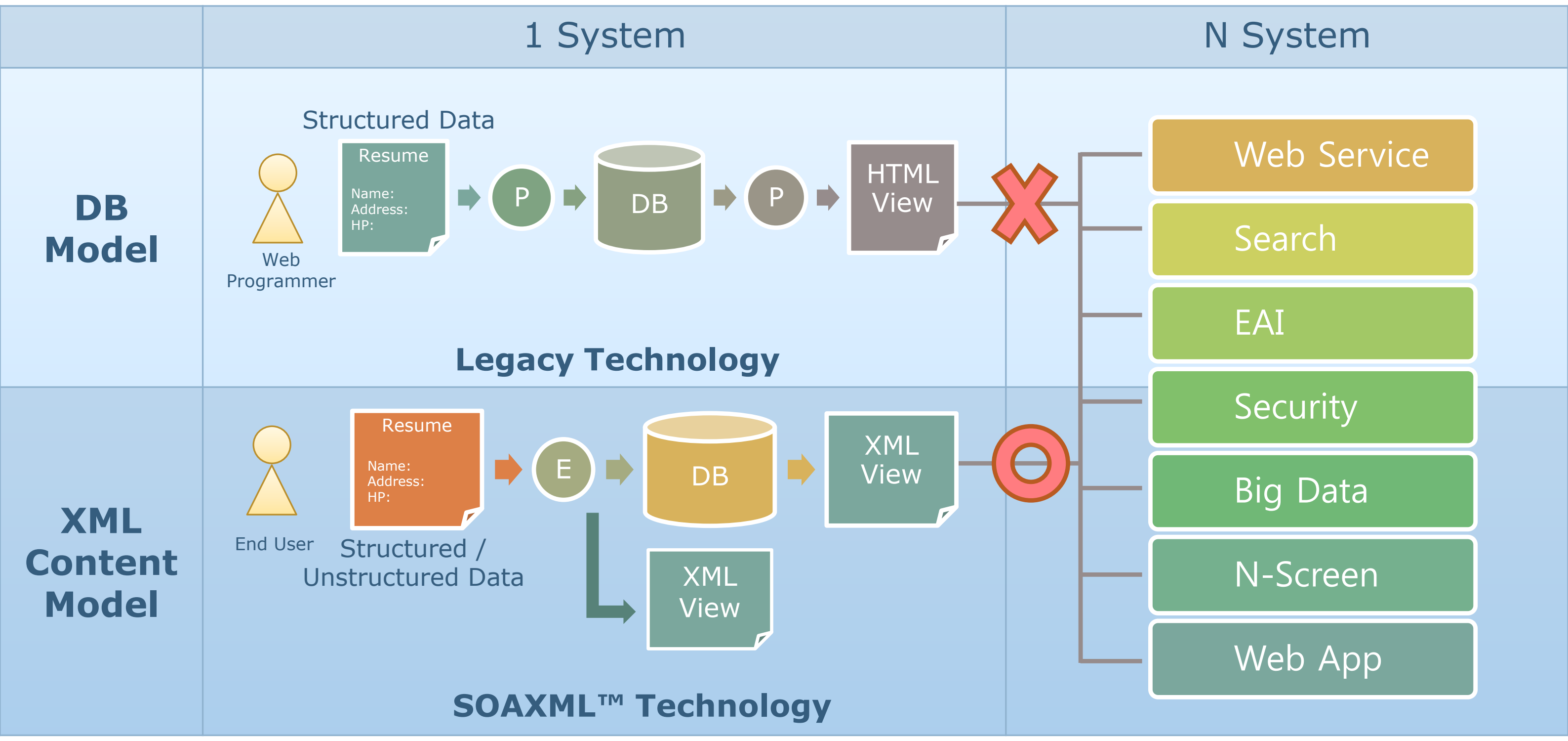


Intuitive web-based XML-editor

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

www.xeditor.com

" XML is not dead, but alive"



3K SOFTWARE
XML SOLUTION PROVIDER

www.3ksoftware.com
www.xmlidc.com
info@3ksoftware.com

3KsoftwareUSA, Inc.
Suite 375,
3 Twins Dolphin Drive,
Redwood City, CA, 94065

3Ksoft
3F, Hyundai Topics Bldg,
44-3, Bangi-dong,
Songpa-gu, Seoul, Korea

<oXygen/>®
XML Editor

XML Authoring

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

XML Development

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

www.oxygenxml.com

MarkLogician

MarkLogic Consultancy Services

marklogician.com

General Information

Date

Saturday, June 7th, 2014
Sunday, June 8th, 2014

Location

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

Produced By

XML London (http://xmllondon.com)

Sponsors

Gold Sponsor

- MarkLogic - http://www.marklogic.com

Silver Sponsors

- Xeditor - http://www.xeditor.com
- 3Ksoftware - http://www.3ksoftware.com
- oXygen - http://www.oxygenxml.com

Bronze Sponsors

- Antenna House - http://www.antennahouse.com
- Saxonica - http://www.saxonica.com

Preface

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

This is the second international XML conference to be held in London for XML Developers – Worldwide, Semantic Web and Linked Data enthusiasts, Managers / Decision Makers and Markup Enthusiasts.

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

The conference is taking place on the 7th and 8th June 2014 at the Faculty of Engineering Sciences (Roberts Building) which is part of University College London (UCL). The conference dinner and the XML London 2014 DemoJam is being held in the Jeremy Bentham Room at UCL, London.

The conference is held annually using the same format, with XML London 2015 taking place in June next year.

— Charles Foster
Chairman, XML London

Benchmarking XSLT Performance

Michael Kay

Saxonica

Debbie Lockett

Saxonica

Abstract

This paper presents a new benchmarking framework for XSLT. The project, called XT-Speedo[1]*, is open source and we hope that it will attract a community of developers. The tangible deliverable consists of a set of test material, a set of test drivers for various XSLT processors, and tools for analyzing the test results. Underpinning these deliverables is a methodology and set of measurement objectives that influence the design and selection of material for the test suite, which are also described in this paper.*

1. Objectives and Motivation

Performance of XSLT is, of course, important, though we need to qualify that by pointing out that performance is not the only thing that matters.

For Saxonica as a developer of one of the leading XSLT engines, performance is not actually our number one objective. Our first objective is standards conformance; the second is usability, and performance comes third. This means that we will (almost!) never sacrifice standards conformance in order to achieve improved performance, and we are prepared to take a performance hit in order to improve usability, for example by maintaining extra run-time information that is needed only for diagnostics when things go wrong.

We choose these priorities because we think this is what the market wants. We are probably rather obsessive about corner cases when it comes to standards conformance, but being obsessive about cases that most users don't care about means that we almost always get things right in the more common cases where users care a lot. Standards conformance is typically a major objective for vendors who can't take their place in the market for granted, and for Saxonica, achieving standards conformance was what gave us a place at the top table of XSLT vendors alongside the likes of Microsoft and IBM.

As for usability, we believe that far more users notice sub-standard usability than notice sub-standard performance. Most users only care that performance is good enough, not that it is the best available. If one processor runs a transformation in 0.1s and another does the same work in 0.2s, most users won't notice the difference. They are much more likely to end up using your product because they discover, from experience, that its error messages are more helpful.

But even though performance is our third priority, it's still extremely important. Saxonica has users running some seriously impressive workloads, and when we get things wrong, they notice.

In measuring performance, our main objective is to avoid regression between successive releases. Such regression is probably the most obvious source of complaints; the last thing users want if they move forward to a new release is to find that their particular workload runs more slowly. Although we have always included some performance tests in our standard quality assurance checklist before release, experience has shown that these are inadequate, and too many mistakes slip through.

Another significant objective is to be able to test the impact of product changes. When we introduce a change (perhaps a new optimization) that is designed to improve performance,we will often do some ad-hoc tests to ensure that the particular test case it is tackling shows the expected improvement. But assessing the overall impact of the change is much more difficult.

[1] *XT-Speedo - https://github.com/Saxonica/XT-Speedo/*

doi:10.14337/XMLLondon14.Kay01

Some readers may be surprised that we do very little competitive benchmarking of our own product against products from other vendors. We did more of this in the early days, before Saxon became well established. One explanation is that the primary reason people adopt Saxon has always been that they want access to XSLT 2.0, and are switching from a product that only supports XSLT 1.0. In that situation, the only thing they want to be sure of is that the change will not cause an unacceptable performance hit. We've had lots of positive feedback from users making this transition, and very little negative feedback, so we haven't felt any pressure to improve our competitive position, although we have always been aware that some of the competitors' products have excellent performance.

As regard competitive performance, it's worth emphasizing that this is not a race in which the winner takes all. Firstly, speed is not a one-dimensional metric, so there will never be a single winner regardless what you choose to measure. Secondly, being within 5% of the leader is good enough for all practical purposes, since if two products differ only by 5% in performance, then users will choose between the products based on other factors. If a user has a performance problem, it's most unlikely that a 5% improvement will solve it.

2. Previous work

There have been previous benchmarking environments for XSLT.

An early attempt was the XSLTMark benchmark from Datapower, which later became part of IBM. Datapower offered this as a free download from their web site for a number of years, but it disappeared at around the time of the IBM acquisition. The licensing terms were unspecified, so although we (and no doubt others) have continued to use this benchmark in-house, we have no authority to make it public. XSLTMark suffered a common problem when comparing results across different products: different drivers measured different things. For some products, the cost of compiling the stylesheet was included in the execution cost, for others it was discounted. This made product comparisons fairly useless, but it was still a useful tool for comparing successive releases of the same product.[1][2]

XSLT processors have come on a long way since 2001, but it's very hard to find any more recent data that compares their performance.

Another benchmarking effort that appeared on the web and then disappeared leaving very little trace was Bumblebee. The main problem with this effort was that the drivers were not open source, and there was insufficient information provided to create your own drivers. We used it in Saxonica for a while, but when we found that we couldn't update or tweak the drivers to experiment with different settings, we abandoned it.

Sarvega published results of a benchmarking study in 2003, but the results were only available on a commercial basis, and the tools needed to reproduce the results were not made available at all.

A more recent effort is XSLTMark II[3], produced by Viktor Mašíček as an M.Sc thesis at Charles University, Prague. Mašíček is concerned to measure more than just performance, but although he recognizes the importance of other qualities such as usability, he does not attempt any scientific measurement. He also on occasions fails to distinguish correctness from usability, for example when he commends XSLT 1.0 processors that reject XSLT 2.0 constructs rather than ignoring them, which might well be the right thing to do from a usability perspective, but happens to be non-conformant with the XSLT 1.0 specification.

On performance, Mašíček's reported results suffer from a failure to distinguish the different factors that contribute to execution time. In the case of Java processors, he makes insufficient effort to discount the effects of Java VM warm-up time; for example, he reports Saxon-HE 9.4 as running three times slower than Saxon 6.5, an effect which can only be explained by the fact that Saxon-HE is larger and therefore takes longer to load. For some workloads this start-up cost matters to users, but for production web sites running thousands of transformations per hour, as well as for one-off transformations of very large documents, start-up cost is irrelevant. The same arguments apply to the cost of stylesheet compilation; Mašíček makes no attempt to distinguish compilation time from execution time, but for many workloads, compilation time can be ignored.

Another problem with Mašíček's benchmark is that he runs every processor in the same environment (PHP). For processors that are not designed to run in this environment, this creates an artificial overhead. For example, to invoke Java processors he uses what is essentially a command-line invocation. The overhead of invoking a Java processor in this way swamps the actual transformation cost, so the measurements are entirely untypical of what can be achieved in a native Java environment.

[1] A description of the XSLTMark benchmark, including an acknowledgement of the problems in using it for cross-product comparisons http://www.xml.com/pub/a/2001/03/28/xsltmark/index.html

[2] A set of XSLTMark results from 2001 http://www.xml.com/pub/a/2001/03/28/xsltmark/results.html

[3] XSLTMark II http://xsltbenchmarking.masicek.net

Nevertheless Mašíček's work is valuable, in particular the collection of stylesheets that he has collected, and our work builds on this.

Other relevant work includes benchmarks for XQuery and XPath. For XQuery the XMark benchmark[1] is the best known. This includes a test data generator which can be used to create data files of different sizes, which therefore enables measurement of how query performance scales with data size. This is particularly useful to enable comparison of strategies for join optimization in different products. We have used this benchmarking framework extensively in Saxonica, and have adapted some of the 20 queries to XSLT, but they are not very typical of real-world XSLT workloads and this limits their usefulness.

A more recent paper from 2006 surveys XQuery benchmarks[2]. This paper contains much useful discussion of the characteristics of a good benchmark and the kind of information it can yield if analysed intelligently; in particular, the importance of determining the way in which performance varies (for example, with document size or query complexity) rather than collecting simple numbers.

3. The design of the XT-Speedo benchmark

The XT-Speedo benchmark has been designed to measure the performance of different products for a broad range of test transformations. In particular we have chosen to measure the time for three processes (where possible): compiling the stylesheet, file to file transform, and tree to tree transform. Since different vendors may have varying priorities, and since it is interesting in itself to see the difference in performance for these different parts of the transform process, we considered taking these three measurements to be useful.

We have produced XT-Speedo benchmark packages on the Java and .NET platforms, and in C/C++, to run different test drivers for the different products which are available in different environments. For each environment, the packages each contain a class to run the benchmark tests (called Speedo on Java, and RunSpeedo on .NET), and an abstract class called IDriver, which is subclassed for each product-under-test. The IDriver interface defines methods to compile a stylesheet, to load a schema, to build a source document, and to run tree-to-tree or file-to-file transformations.

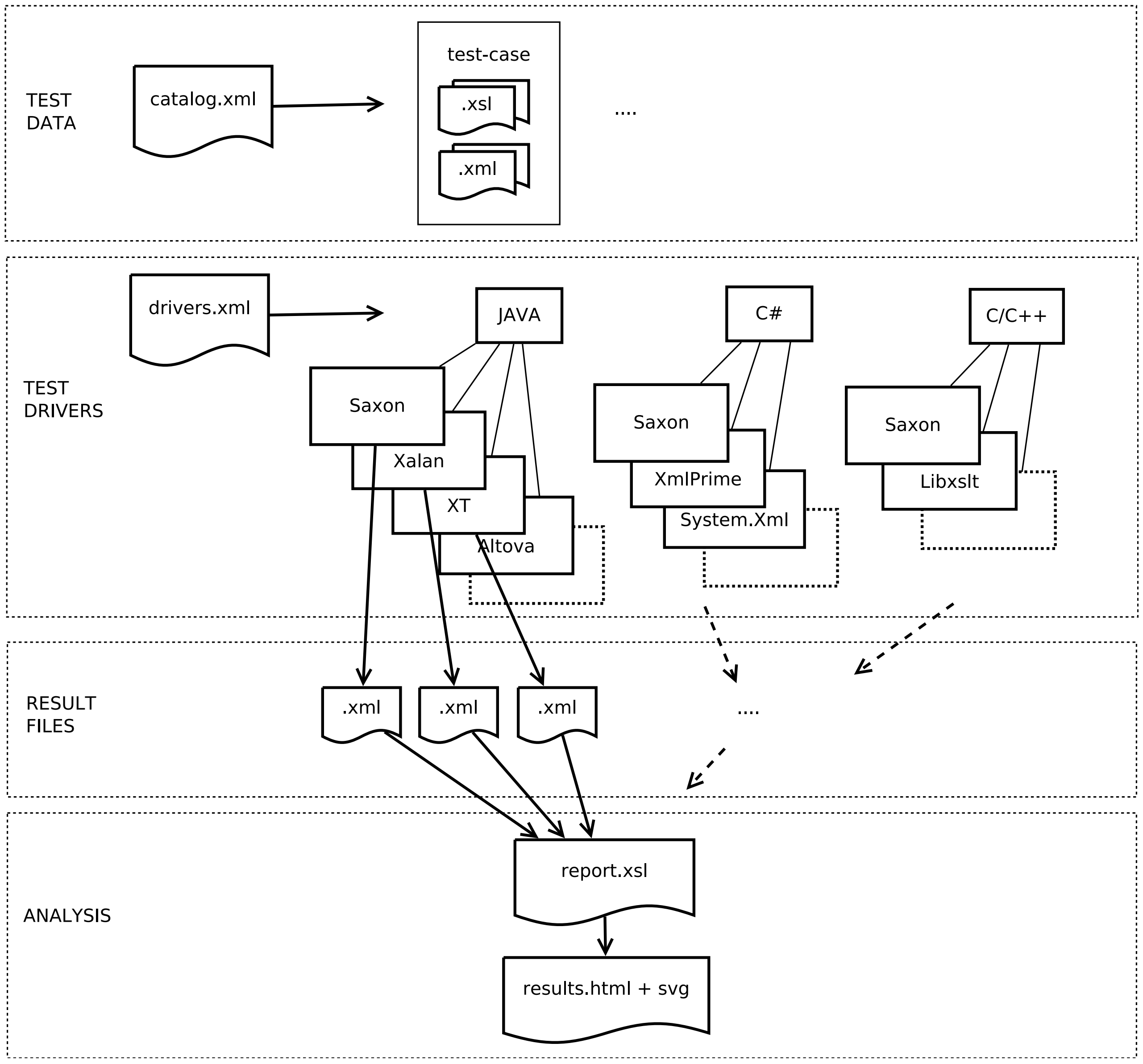
Not every product supports all these options. Some, obviously, are not schema-aware. Some, such as Altova's RaptorXML, do not provide an explicit interface to compile the stylesheet (perhaps they rely on caching instead). Some, such as James Clark's XT, do not separate tree construction from transformation. A further complication is that we want to compare the performance of all processors when running XSLT 1.0 tests, but we are also interested in XSLT 2.0 and XSLT 3.0 performance, so we have to accommodate the fact that for different processors, we may have different subsets of the measurements, available over different subsets of the tests. We therefore use XML-based catalog files to control which tests are run, using which processors.

The design of the benchmark is illustrated by the following schematic:

[1] XMark benchmark http://www.xml-benchmark.org/index.html

[2] XQuery benchmarks survey http://leo.saclay.inria.fr/events/EXPDB2006/PAPERS/Afanasiev.pdf

Figure 1. Architecture diagram



The main command line input for the Speedo benchmark (i.e. input for the 'run' method of the Speedo class) is the tests catalog file and the drivers catalog file. An option is also provided to supply the location of the output directory for result files. Further options are provided to select which test cases from the catalog to use — primarily by providing a regular expression name pattern, but also for example by choosing to skip tests which are slow.

The tests catalog ("catalog.xml") identifies a collection of test cases, gleaned from various sources, plus some newly created ones. Each test case is a particular transformation for the product to process. The catalog contains a description of the test transformation, links to the source XML file and XSL stylesheet (and in some cases, an XSD schema), and an XPath assertion to be applied to the output of the transformation to check the results are plausible. (It's not a primary aim of this exercise to check the conformance of processors to the specification; but we want to weed out results that are wildly out, especially cases where the processor fails to perform the transformation at all.)

The drivers catalog ("drivers.xml") contains the driver data, including: name, implementation class, implementation language, XSLT version. This is where initialization option settings and test run options can be added. In the case of Saxon, we might have several entries in the drivers catalog that run Saxon with different configuration options (optimization on or off, bytecode generation on or off, and so on), allowing us to assess the impact of these configuration switches. The drivers catalog can also indicate that particular tests should not be run with a particular driver (because they are known to crash, for example, or because they are excessively slow.)

Because different processors run in different environments, collecting a full set of data for all processors requires more than one program run. As a minimum, there will be three runs, one for Java processors, one for .NET, and one for C/C++. But if we want to measure different versions of Saxon, or performance on different hardware or operating systems,then additional runs will be needed. (We have experimented with a mechanism that calibrates the hardware speed and adjusts performance measurements to compensate for differences. However, this mechanism is not fully operational.)

Each execution of the benchmark runs all the selected tests from the test catalog (as selected by the specified configuration), and produces one XML results document per driver. For each test case, measurements are taken for the times (in milliseconds) to perform three processes: compiling the stylesheet, file to file transform, and tree to tree transform. Each test case is run multiple times (by default set at 20) and the average times for these processes are taken, in order to eliminate warm-up time and aberrations caused by activities such as garbage collection. (For Java in particular, hotspot compilation causes dramatic improvements in execution time the more often the code is run, with performance often stabilizing only after a minute or so.) These average process times are recorded in the result file, indexed by the test cases, where the level of success of the test run is also recorded - 'success' if the run is fine, 'wrong answer' if transforms take place but the assertion test fails (so the transformation result is not as expected), and 'failure' if the transformation fails at some point.

A selected subset of these result files can then be collated using the "report.xsl" XSLT stylesheet. Using the results XML files as input, this stylesheet produces HTML documents (including SVG graphics) to view the results. The results tables give times relative to a selected baseline driver (chosen in the drivers catalog). The main overview page contains, for each driver, a measure of the overall performance relative to the baseline for each of our three processes: file to file transform, tree to tree transform, and stylesheet compile. The formatted report for each driver contains tables with rows for each test case, giving the relative times, and actual times, for the three processes.

As our "bottom-line" metric we use the sum of the process times (over all tests) for the driver, relative to the sum for the baseline driver. This of course gives greater weighting to tests which take longer; a different choice of computation could well give a different picture. We also give the minimum and maximum values of the relative times for the individual tests, to give an idea of the spread. We fully recognize that this is highly arbitrary; there are other ways of doing the aggregation that would give different results. XSLTMark, for example, divides execution time by source document size to give a transformation speed measured in bytes per second, and averages across these speeds. In some cases, as we will see, relative performance of different processors varies substantially from one test to another, and because our test collection makes no serious attempt to be representative of any real workload, an average across all the tests can fairly be dismissed as meaningless. In defence, users of the benchmark are free to substitute a different set of tests that reflects their own choice of workload more accurately.

4. Test Data

The selection of data files and stylesheets used in the benchmark should not be regarded as being fixed in concrete. XT-Speedo is intended as a framework for measuring XSLT performance, not primarily as a set of test programs which claim any kind of canonical status. The variability of results across the different test data sets often provides more information than any aggregate numbers. The data included in the benchmark is a motley collection, including some things that just happened to be available (for example, files from the original Datapower XSLTMark benchmark), some that we wrote specially because we wanted to investigate a particular area of performance, some that we have used in the past to study particular performance issues reported by Saxonica customers, and also a translation of the XMark XQuery benchmark, allowing us to see how XSLT and XQuery performance compare. The XMark data is particularly useful because it allows one to study how performance varies as a function of the size of the source data set.

Any attempts to aggregate results over all the tests are inevitably flawed. While a figure that averages performance across a range of different tasks is likely to be more reliable than a figure for one task alone, it would be quite wrong to assume that the tests in this benchmark collection are representative of any real production workload. Although they are nearly all real programs designed to perform (or at least emulate) a useful task rather than to stress obscure corners of the processor, some of them perform rather untypical tasks, such as parsing XPath expressions.

It is therefore quite legitimate, and positively encouraged, to run the XT-Speedo benchmark with different data files that better characterize the workload for which performance data is required. Unlike some classic industry benchmarks such as TPC, we have no aspiration to define a performance metric that vendors can publish on billboards to proclaim that their product is 32.87% faster than the competition. Rather, the benchmark is a resource that anyone can use to compare different workloads in different environments in any way that suits their purposes.

5. The Problem of Bias

We are acutely aware that our results are not impartial. We know that our own motivations are divided between wanting to know the truth about how our product rates against the competition, and wanting our own product to perform well.

The problem of bias arises from several sources: requirements, expertise and motivation.

- *Requirements:* in designing the benchmark, we are choosing what to measure, and the metrics we choose reflect our assumptions about what we think is important. For example, we consider compile-time performance much less important than run-time performance. But others might have different priorities. Similarly, all our measurements focus on latency rather than throughput (the time to execute transformations in a single thread). We quickly found that one particular processor, Altova RaptorXML, fares very badly on this metric, because it is designed to execute in an HTTP server environment and is clearly optimized for throughput rather than latency. The fact that it scores very badly on our measurements does not mean it would score equally badly if we chose different metrics.
- *Expertise:* we know how to get the best possible performance out of our own processor, but we have far less knowledge of the products of our competitors. We've seen third-party benchmarks that ran Saxon in hideously inefficient ways (for example, taking the input from a DOM tree, which can increase transformation time by a factor of ten), and we know that we are at risk of making equally bad choices when running other products that we are less familiar with.
- *Motivation:* we naturally want to get the best possible results for our own product, so if the results don't look good, we will instinctively try again with different settings. We don't have the same motivation for other products, so we are less likely to make the effort. To take an example of this effect, when we compared Saxon/C against libxml our first attempts showed Saxon/C in a very poor light. We naturally investigated, and found a gross error in the way the measurements were being computed. Can we honestly say that we would have investigated as thoroughly if the results had been the other way around?

The bias is there despite our best intentions. We want good data on our competitors' products; we don't want to deceive ourselves. Our best defence against bias is to make the benchmark open source. Our hope is that we will get contributions from others whose bias is different, in particular, who will apply the same diligence to other products as we apply to our own. Meanwhile, however, the biased results are the only ones available.

Because we know there is bias, we refrain from publishing detailed data for our competitors' products in this paper. The results are available on the web site, where they can be corrected if they turn out to be wrong. They will also be presented in the conference, but as a snapshot of current results, not as part of the permanent record.

The problem of bias arises far less when we are comparing our own product, Saxon, running in different versions and configurations. As explained earlier, this is in fact our primary motivation for producing the benchmark. So in the next section these results can be seen as more impartial than the competitive rankings.

6. Selected Results

In this section we will present some of the results we have obtained by running the benchmark, and our analysis of these results. We focus on five particular comparisons: an overall comparison of all processors on the Java platform (all of which, with the exception of Saxon, are XSLT 1.0 processors); a comparison of Saxon on the Java and .NET platforms; a comparison of Saxon with XMLPrime, this being the only other XSLT 2.0 processor we were able to study; a comparison of Saxon 9.5 with a current development snapshot of the forthcoming Saxon 9.6 release.; and a comparison of the new Saxon/C processor with libxslt.

6.1. Ranking of Java Processors

A number of XSLT processors have been developed for the Java platform: as well as Saxon, there are several versions of Xalan, including the XSLTC processor which was developed separately but is now bundled with the Xalan distribution; there is James Clark's original XT processor; there is the no-longer-available jd.xslt, and there is IBM's commercial Websphere processor. Of these, Saxon and Websphere are the only two processors that support XSLT 2.0, and the only two that are still actively developed. For commercial reasons we have not been able to include Websphere in our study. A comparative study of Java processors is therefore confined to XSLT 1.0, and the interesting question for us is how Saxon stacks up against products that have been around and stable for many years.

Here are the XT-Speedo results we are currently getting, using Saxon EE 9.5 as the baseline (recall that we do not get tree to tree transform times for XT).

Table 1. Results overview table for Java drivers

	Times relative to SaxonEE-9.5-J driver (smaller values represent faster times)		
Driver	File to file transform	Tree to tree transform	Stylesheet compile
Saxon-6	1.108 min = 0.271, max = 4.33	3.915 min = 0.178, max = 429.71	0.204 min = 0.081, max = 0.779
SaxonHE-9.5-J	1.076 min = 0.512, max = 3.329	1.279 min = 0.325, max = 98.844	0.212 min = 0.083, max = 1.802
Xalan	2.452 min = 0.467, max = 13.379	8.911 min = 0.37, max = 568.723	0.283 min = 0.1, max = 1.284
XSLTC	0.989 min = 0.273, max = 3.407	3.142 min = 0.182, max = 257.989	0.544 min = 0.203, max = 3.195
XT	1.398 min = 0.336, max = 7.717	NaN min = NaN, max = NaN	0.23 min = 0.088, max = 0.886

The tree-to-tree transformation times shown here illustrate the difficulty of getting good measurements on the Java platform. For all processors, the "max" figure indicates the presence of outliers in the results that create a completely distorted bottom line. If these rogue results are excluded, the figures end up being much closer to the file-to-file timings. So we'll concentrate on the file-to-file numbers as they appear to show a more regular picture.

These figures show Saxon-EE performing 7% faster than Saxon-HE on average, which is not as large a margin as we would like given the investment we have made in features such as optimization and byte-code generation, but perhaps reflects that these advanced techniques make little impression on straightforward transformations which dominate the test suite. The 10% edge over the old Saxon 6 processor (which implemented XSLT 1.0 only) is also a satisfactory outcome. In fact these figures mask the fact that there are a few transform times where Saxon-EE dramatically outperforms the other processors because of the way in which it optimizes joins: for the test xmark-q8-4 Saxon-EE is almost 100 times faster than Saxon-HE (most processors have quadratic performance on this test, which Saxon-EE optimization reduces to near-linear).

James Clark's original XT processor is now of largely historic interest, but in the early years it was noted for its lightning-fast speed, so it is good to note that we are now 40% faster.

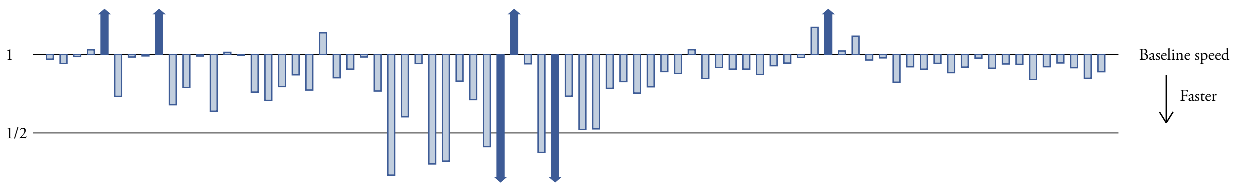
Saxon's very significant advantage over the interpretive version of Xalan should not surprise anyone who has compared the two. The fact that Xalan, being the default XSLT processor in Java, is both the most widely-used and the slowest of these products by a significant margin, tends to reinforce the message at the beginning of this paper that coming first in the performance race brings no guarantee of market leadership.

The only product ahead of Saxon-EE is the XSLTC processor (which is bundled with Xalan). This processor makes heavy use of bytecode generation and the results appear to demonstrate that there are still advances to be made in this area. (Saxon-EE also uses bytecode generation, but primarily for the XPath part of the processing. Most of our measurements of the effect of bytecode generation have been with XQuery, where we generally record a boost of around 25%, but with wide variations. It is not surprising that the speed-up we get for XSLT should be lower.)

Let's take a closer look at the comparison of Saxon-EE with XSLTC results (see charts below). Here we see in more detail that for file to file transform, in the majority of tests XSLTC is just a few percentage points better than Saxon-EE, with XSLTC generally slightly faster (with just a few exceptions). For many of these tests, especially the XMark queries which dominate the right-hand half of the chart, the actual performance for file-to-file transformation is dominated by parsing and serialization costs, and it appears to be in these areas that XSLTC has the edge.

Tests whose results are outside the 95th percentile range are shown with an arrow to indicate they are off the scale. The actual numbers are available in the detailed results listings. In this particular example, the outliers are not extreme; for all tests the ratio between XSLTC speed and Saxon-EE speed is somewhere between 0.25 and 4.

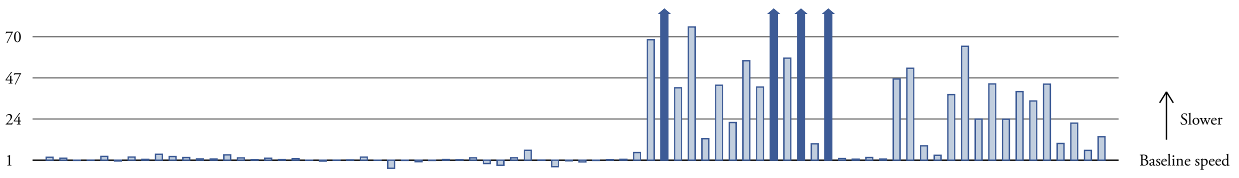
Figure 2. XSLTC file to file transform speeds relative to SaxonEE-9.5-J



For tree-to-tree transformation we see a very different picture (below). For these transformations the times for XSLTC are generally close to Saxon or a little slower on the left-hand part of the chart where source documents are mainly rather small, but on the right-hand side, where most of the source documents are 1-4Mb in size, XSLTC is significantly slower than Saxon-EE. What isn't immediately obvious from these charts is that for Saxon, the tree-to-tree time for the larger source documents is a tiny fraction of the file-to-file time (in one typical example, 0.24ms rather than 20.4ms), but in the XSLTC case the timings for the two scenarios are much closer

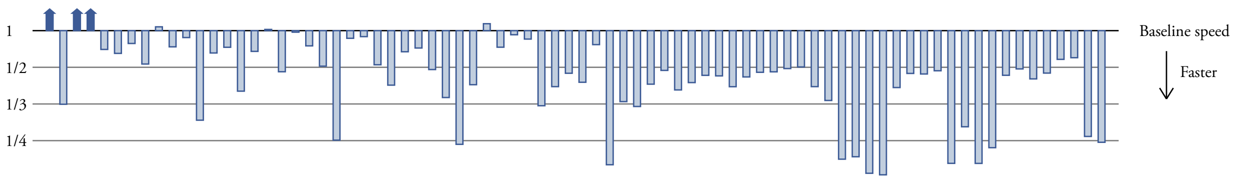
(10.1ms compared with 16.2ms). We suspect that we are running XSLTC sub-optimally here by providing a DOM as input. Perhaps it is not using the source tree that we supply directly, but rebuilding it internally into its own format.

Figure 3. XSLTC tree to tree transform speeds relative to SaxonEE-9.5-J



Our final metric is for stylesheet compile time. Here the figures are fairly uniform across all tests, with XSLTC compiling in around half the time of Saxon-EE on average:

Figure 4. XSLTC stylesheet compile speeds relative to SaxonEE-9.5-J



Both Saxon-EE and XSLTC compile to bytecode, so it is not surprising that both are significantly slower at compile time than the products that are pure interpreters. It is noteworthy that Saxon-EE takes significantly longer to compile the stylesheet than all the other processors. We could argue that this is by design; we deliberately do as much work as possible at compile time in order to improve run-time execution speed. On the other hand, there are workloads (the DocBook rendering of this conference paper is an example) where compiling the stylesheet takes longer than the actual transformation, and there is definitely an opportunity here for Saxon to do better.

The conclusion we can draw from these results is that while Saxon is not always the fastest, it performs well overall. In particular, for anyone wanting to move forward to XSLT 2.0 for the functionality and productivity benefits it offers, or who is attracted to Saxon because the product is actively developed and supported, performance is not an obstacle. For some workloads, users have seen significant performance benefits by moving to Saxon, but as the numbers show, this cannot be expected to apply in every case.

The other apparent result, subject to confirmation, is that the area where Saxon-EE has most improvement potential is in parsing and serialization, not in transformation proper. There are a great many workloads where XML parsing (of the input) and serialization (of the output) dominate the actual transformation time.

6.2. Comparing Saxon on Java with Saxon on .NET

Saxon on .NET starts with a disadvantage: the product is written in Java, and then cross-compiled using the IKVMC compiler to the IL code supported on the .NET platform. This inevitably introduces a performance penalty.

The question for some time has been how large this penalty is, and we have had conflicting reports on this over the years. Sometimes we see an overhead of around 25%, but sometimes the .NET performance is reported to be five times slower.

Here are the XT-Speedo results we are currently getting: File to file transform relative time average 3.829 (min 1.136, max 8.716), tree to tree transform relative time average 3.598 (min 0.239, max 8.938), stylesheet compile relative time average 2.386 (min 0.168, max 3.454).

Figure 5. SaxonHE-9.5-.NET file to file transform speeds relative to SaxonHE-9.5-J

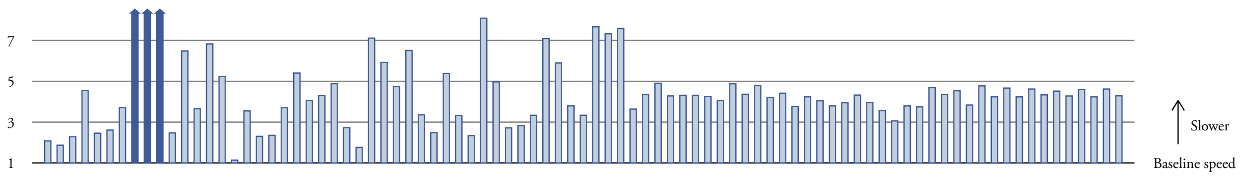


Figure 6. SaxonHE-9.5-.NET tree to tree transform speeds relative to SaxonHE-9.5-J

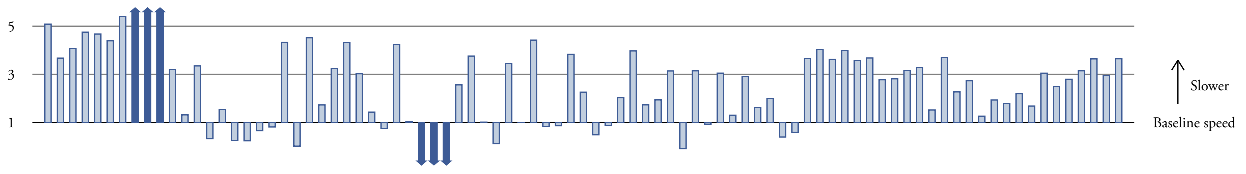
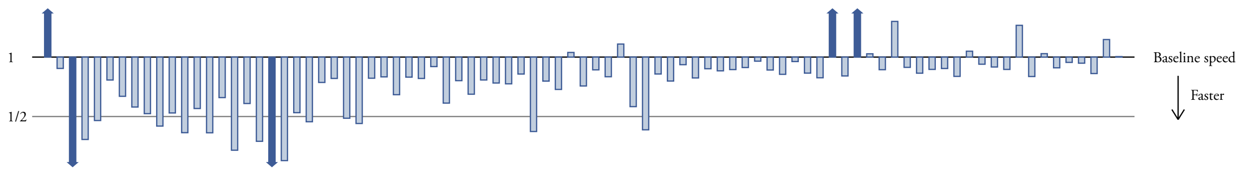


Figure 7. SaxonHE-9.5-.NET stylesheet compile speeds relative to SaxonHE-9.5-J



It is clear that generally Saxon on .NET is indeed running transforms 3 to 4 times slower than on Java, with some variation for different tests. Perhaps surprisingly, Saxon on .NET sometimes performs tree to tree transforms faster than on Java. By looking at the table of results (not shown here) we see that .NET speeds for tree to tree transform are only ever faster for transforms which are very quick - those which take less than 2ms (but .NET is not uniformly faster for these). Generally performance worsens for longer transforms, but we may note that in general the scaled pairs of xmark tests have similar relative times.

The chart for compile times is particularly remarkable, because most of the compilations are actually faster on .NET, but the average is still slower: the explanation for this paradox is that most of the stylesheets are very small, but one test (on the far left of the chart) compiles the DocBook stylesheets, which are much larger than all the others combined. Again the performance ratio seems worse for longer runs. One possibility we need to explore is that we are measuring different things on the two platforms (what are the precise semantics of the instrumentation APIs we are using?), or that the measurements are somehow suffering from rounding errors.

We do not yet fully understand the reasons for the discrepancies in these results. We have established that there are no significant differences in the code path with Saxon, and we know that the overhead imposed by IKVMC is not more than 25% or so. So far, our investigations suggest that the problem lies somewhere in the OpenJDK library. Saxon on .NET uses the OpenJDK java library, cross-compiled to .NET using IKVMC, and the data makes us suspect that there are parts of this library that perform significantly worse than the equivalent library delivered with the Oracle/Sun JDK. We have confirmed this by building Saxon on the Java platform to run with OpenJDK rather than with the Oracle/Sun libraries. Hopefully, armed with these measurements, we can identify a specific cause within the OpenJDK and eliminate it. As always, good measurement data is the prerequisite to solving performance problems, and we now for the first time have that data.

6.3. Comparing Saxon with XMLPrime

For various reasons (which would make an interesting subject for another talk), none of the XSLT 2.0 processors currently on the market are pure open source products. Products from IBM, Intel, and MarkLogic are purely commercial, while those from Saxonica, Altova, and XMLPrime provide limited free versions in one form or another, but offer only commercial licenses for the full product capability. This of course makes product comparisons much more difficult and expensive.

The other XSLT 2.0 processors we have included in our study are Altova's RaptorXML and XMLPrime. In the case of Altova RaptorXML, the product architecture is so different that the figures we obtained were not meaningful to compare; each transformation requires an HTTP request, and our performance data was dominated by the costs of these requests. No doubt much better figures could be obtained for Altova if we did the measurements a different way, but for the present we have discarded the numbers as not useful.

XMLPrime, on the other hand, has a very similar architecture to Saxon; indeed, a cursory glance at its structure shows that it was strongly influenced by Saxon's design. So measurements here should be useful.

The most interesting result here is to show relative speeds for each test case. We see that the pictures for file to file and tree to tree are closely related. In general, XmlPrime is running just a little slower than Saxon EE, but it is sometimes faster. There are just a few cases where XmlPrime is much (more than 5 times) slower than Saxon, and here we see the difference for both file to file and tree to tree transform times. These cases are all among the tests which take longest, and so are meaningful. In contrast, the cases for which XmlPrime is much faster than Saxon are all very quick tests, so we may consider these numbers to be less reliable and meaningful - the fact that these cases are different for file to file and tree to tree transforms, where we have already said that the results correlate strongly, backs this up.

Because Saxon is faster on some tests, while XMLPrime is faster on others, any "bottom line" comparison of the two products is highly sensitive to the choice of test material, and to the way in which the results for different tests are aggregated. The formula we use for aggregation shows file to file transform relative time average 4.581 (min 0.266, max 189.62), tree to tree transform relative time average 9.753 (min 0.29, max 469.095), stylesheet compile relative time average 1.18 (min 0.218, max 21.681). But from the charts, we see that these figures would change greatly if a few anomalous cases were removed. If outliers are discarded, XmlPrime is on average perhaps about 2 times slower that Saxon on transformation time.

We see more variation in the stylesheet compile times, and here more generally XmlPrime performs a little faster. There are two anomalous cases, for which XmlPrime is more than 20 times slower - these are cases that also showed far worse transform times.

Figure 8. XmlPrime file to file transform speeds relative to SaxonEE-9.5-J

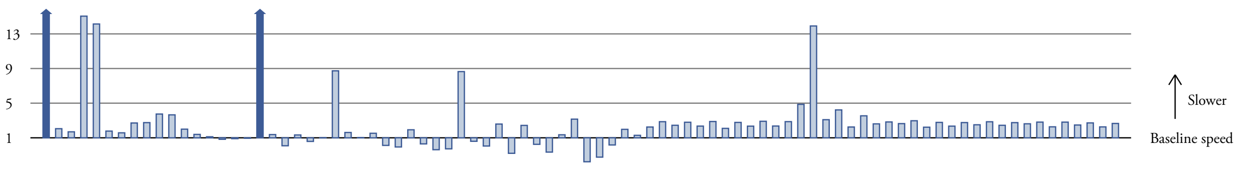


Figure 9. XmlPrime tree to tree transform speeds relative to SaxonEE-9.5-J

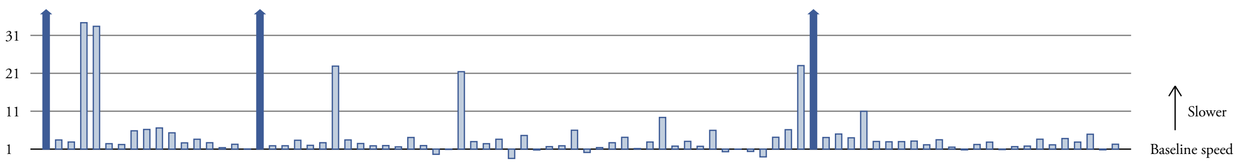
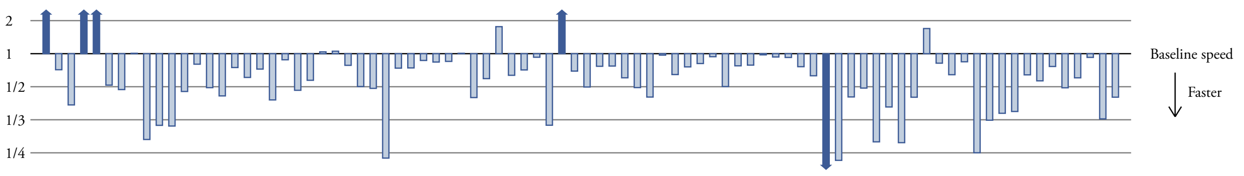


Figure 10. XmlPrime stylesheet compile speeds relative to SaxonEE-9.5-J



The most appropriate way of using these results is not to compute a crude ranking, but to try to understand where each product is stronger and where it is weaker. However, isolating the features of the individual tests to achieve such an understanding is not an easy task.

6.4. Comparing Saxon 9.5 with Saxon 9.6

As already stated, a key aim in writing this benchmark was to enable us to avoid regression when shipping new Saxon releases. Measuring Saxon 9.6 (currently under development) with the current 9.5 release is therefore particularly relevant.

Looking across all tests, this is the data we are currently seeing:

Figure 11. SaxonEE-9.6 file to file transform speeds relative to SaxonEE-9.5

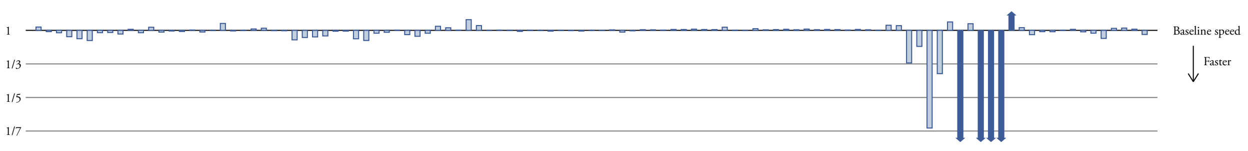


Figure 12. SaxonEE-9.6 tree to tree transform speeds relative to SaxonEE-9.5

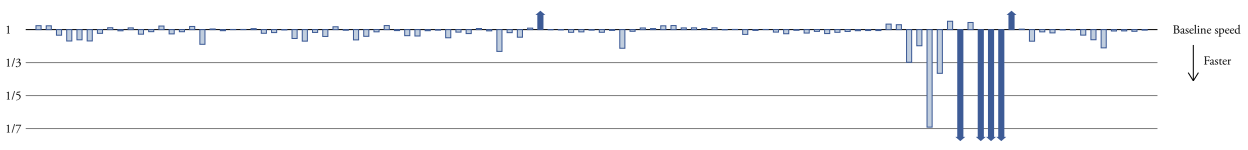
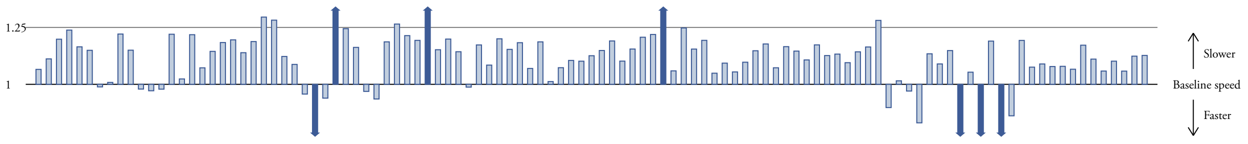


Figure 13. SaxonEE-9.6 stylesheet compile speeds relative to SaxonEE-9.5



What we would expect to find here is that for the majority of tests, the performance is much the same between the two releases, but for a minority of tests, the performance may benefit from deliberate enhancements in particular areas, or it may reveal performance bugs that we need to address before shipping the final product.

There is a lot of noise in the results. There's no reason at all why the performance ratio between the two releases should be different for tree-to-tree transforms than for file-to-file transforms. The fact that some of the ratios are significantly different can be taken as a measure of the inaccuracies that arise during measurement of Java performance, caused (we believe) by the failure to suppress unrelated activity on the system under test, for example Java garbage collection, network traffic or virus checking.

Nevertheless, the overall picture is good. Most tests are showing a performance ratio close to one, and a cluster of tests are showing a substantial improvement.

This cluster of tests was specifically designed to assess the effectivness of a redesign in 9.6 in the implementation of maps. Maps are a new feature in XSLT 3.0, providing a data structure akin to what some languages call "dictionaries" or "associative arrays": a set of key-value pairs providing efficient access to the value associated with any key. As befits a functional language, the maps in XSLT 3.0 are immutable, and herein lies the performance challenge. In Saxon 9.5, the implementation uses a layering of hash maps and deltas, with deltas being absorbed and consolidated when they reach a certain size. In Saxon 9.6, this has been replaced with a hash trie, similar to the structure used to implement immutable maps in Scala.

To ensure that the new implementation performs better than the old, we wrote a number of tests specifically focused on creating, using, and modifying maps. (We can note in passing that the existence of these tests immediately means that our aggregate performance results attach disproportionate importance to this area.)

Our first results from these tests were very discouraging: they showed the new code running five times slower than the old, and we were almost ready to discard it. However, closer study revealed the reason for the discrepancy: the implementation was caching data relating to the types of the keys and values held in the map, and this cache was not being maintained correctly. Fixing this problem gave us new data which showed map construction and retrieval taking a very similar amount of time to the old release, addition of new entries being slightly faster, and removal of existing entries dramatically faster. Sufficient evidence to justify accepting the new code into the release.

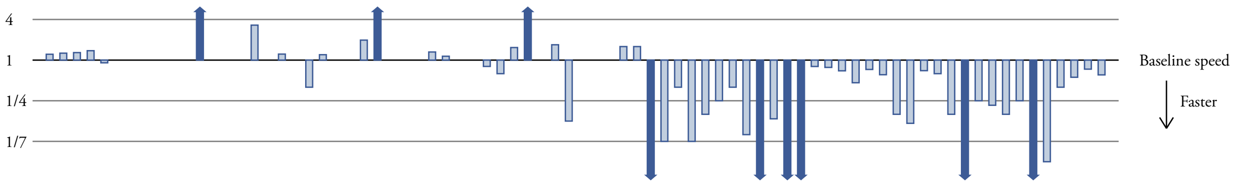
Drilling down even further, we can see variation between the different map tests. For example, test wordmap8 is about 12 times faster on Saxon 9.6, whereas wordmap8a is 1.5 times slower. The two tests are very similar: both construct an index containing all the words in a source document. The difference between the two tests is that wordmap8a, after adding a new entry to the map, counts how many entries are now present in the map using the expression `count(map:keys($map))`. The implication is that in the new data structure, the one thing that runs slower is enumerating the keys present in the map. We can live with this.

The results also show that compile-time performance has got a little worse across all tests in 9.6. This is something we may address before a final release.

6.5. Comparing Saxon/C with libxslt

The newest addition to the Saxonica product stable is Saxon/C: a version of the product issued as a native DLL (or .so) library, suitable for calling from C or C++ applications, together with an interface offering a PHP extension API. This area has for many years been the preserve of the open source libxslt product, which has an excellent reputation, but which (like most of the open source XSLT 1.0 processors) has not been upgraded to XSLT 2.0. Saxonica is aiming to fulfil the demand for an XSLT 2.0 processor in this important space with a version of Saxon that is cross-compiled to native code using the Excelsior JET Java cross-compiler. Clearly the main attraction of Saxon/C to libxslt users will be the ability to take advantage of XSLT 2.0 features, but they will want assurance that performance is adequate.

Figure 14. Saxon/C tree to tree transform speeds relative to libxslt



Our first results comparing Saxon/C with libxslt are shown below. The XT-Speedo driver for libxslt is currently failing many tests when run in file-to-file transform mode, so we present only the tree-to-tree comparison. These show Saxon/C consistently performing better for the tests with larger source documents, and a wide range of results for tests with smaller documents. In the vast majority of cases, however, the speed ratio between the products is between 0.5 and 2.0, so most users are likely to be content. Producing a single metric for the speed ratio is not something we can sensibly do, since it will depend entirely on the selection of tests to run; the only thing we can say with certainty is that Saxon/C consistently performs better for larger source documents.

7. Conclusions

XT-Speedo was written primarily as a resource for use within Saxonica, to enable us to test and compare the performance of our various products. We developed it because the existing performance tests we had been using were woefully inadequate, and because there were too many cases slipping through where, for some particular workload, new Saxon releases showed regression over previous releases despite the release as a whole passing all performance tests.

We have published XT-Speedo as an open source project for a variety of reasons. We want others to be able to reproduce our results and perhaps show us where we have got things wrong or made invalid assumptions. We want others to be able to contribute test data and drivers which we can then benefit from. We also hope that others might be able to take it into areas that we haven't yet tackled, like measuring throughput in a server environment with a concurrent workload.

In this paper we have shown results for a number of performance investigations where we have already found that XT-Speedo gives us new insights into the behaviour of our own products. In some cases the data confirms what we knew and gives us confidence that all is well; in other cases it suggests directions for more detailed investigation, or for remedial action.

We stress again that performance is just one aspect of product quality, one facet that can be used to compare competitive products. It is not the only metric to be used, and is not even our top objective. But we don't want poor performance ever to be a reason for anyone not to use our software. From that perspective, it's not a major concern if Saxon doesn't come at the top of every league table, but the measurements we have taken so far give us confidence that we are in every case within a few percentage points of the leader. More importantly, they tell us where it is possible to do even better.

Streaming Design Patterns or: How I Learned to Stop Worrying and Love the Stream

Abel Braaksma

Abrasoft

Exselt XSLT 3.0 streaming processor

`<abel@exselt.net>`

Abstract

XML and streaming, and more specifically, XSLT and streaming, is often avoided by programmers because they think that streaming is hard. They worry that when they have to rewrite their stylesheets to allow streamed processing, that the stylesheets become less maintainable, (much) harder to develop and that following the Rules on Streamability, in the absence of a good tutorial or book on the subject, is excruciatingly hard and arduous when the only reference they can refer to is the Latest Working Draft on XSLT, section 19.

This paper goes further where a previous paper by me [BRA14][1] left off. This previous paper explains ten rules of thumb for streaming, which will be briefly iterated over in this paper, see Section 4, "Brief overview of the Ten Rules of Thumb of streaming". This paper expands on that by showing streaming refactoring design patterns that turn typical non-streaming XSLT programming scenarios into streaming ones. They can be found in Section 5, "Streaming Design Patterns", the text being specifically geared towards programmers new to streaming.

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

1. Disclaimer

This paper discusses the new rules as they are defined in the most recent public Working Draft, which is, as of this writing, currently in Last Call. In some scenarios, it will apply fixes to this working draft that have been publicly reported on the BugZilla system of the W3C and that have (publicly) received a resolution by the Working Group. The latest version of XSLT 3.0 can be found at [XSLT3] and the current version used for this paper is [XSLWD], which is currently in Last Call WD state[2]. When this paper refers to XPath, it uses the most recent version of XPath 3.0, which is currently in Proposed Recommendation state [XPPR]. The latest version of XPath 3.0 can be found at [XP3]. The related XPath Functions and Operators specifications used is the Proposed Recommendation [FOPR], for which [FO3] holds the latest version.

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

Since the XSLT 3.0 specification reached Last Call status, several issues with it have been reported in [BUGZ] that relate to streaming. Since some resolutions in these public bug reports[3] have a big influence on streamability on situations discussed in this paper, I often refer to these bug reports in the footnotes, so that readers can check the latest status on their resolution. Where possible, I only use bug reports that are closed, which generally means that the resolution mentioned in the bug report is final.

[1] *To prevent over-self-citing and cross-referencing, I have summarized relevant sections of my previous paper, so that this paper can be read on its own, without the requirement to read the previous paper. However, I do recommend reading it, as it serves as a good introduction to streaming in XSLT.*

[2] When a W3C Working Draft reaches *Last Call* it means that it is the last call for people in and outside of the working group to comment on the draft, before it will reach *Proposed Recommendation* status.

[3] The term "bug" is not entirely fair. Both improvement request and bug reports end up in the same BugZilla repository at W3C.

doi:10.14337/XMLLondon14.Braaksma01

2. To stream or not to stream?

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

2.1. Size of the input data tree

The most obvious use-case for using a streaming aproach is the size of data. As a rule of thumb, if the data that needs to be read and written does not fit in memory, you would choose a streaming approach. If your input is an XML file, and you can estimate the size of the output XML file or files, you should add all these estimates together and multiply them by three or four (depending on your processor and the chosen XDM), and add about 200MB to the total (again, depending on the processor). If this total fits into computer memory, you do not need to use streaming. If it does not fit, you should seriously consider using streaming XSLT.

2.2. Intrinsically streamed input data tree

Sometimes, input is streamed by default, that is, it is not all available at once. In such cases, you must use streaming, because only with streamed processing does the processor not require to read the complete input tree at once.

2.3. Streaming output

It is important to distinguish between input streaming and output streaming. The XSLT specification is very thorough when it comes to input streaming and all rules that allow streamability are written in such a way that they also allow output streaming, but whether or not a processor actually does output streaming is not directly required by the specification[1]. Similarly to input streaming, if the output data is too large to fit into memory, you should use a streaming approach.

While processors are required to flush the output buffer in time, you should test the abilities of your processor in this respect, because there is no standardized way of testing this requirement with processors, and streaming is relatively new. If you run into issues with running out of memory and you suspect that the cause is that output flushing is not done in time, you should contact your vendor and ask for a fix.

2.4. Streaming unparsed text

This paper discusses streaming of XML, but XPath 3.0 introduces a new function, `fn:unparsed-text-lines`[2], which takes an external resource as input and parses it line by line. The original intend of that function was to allow unparsed data to be streamed, however the Working Group at some point decided to not formalize this requirement, however, the specification leaves enough room for implementors to allow streamed processing of data read through this function. When your intend is to do streaming of unparsed input, you should check the capabilities of your processor to find out whether it can do streaming using this function.

The analysis of streaming when using unparsed text through this function, is not required, because the result of the function is a sequence of strings, which in streaming terms, is a *grounded* and *motionless* result, because it does not change the read pointer on an input XML tree. This may seem strange, but XPath nor XSLT have expressions that allow you to go from one string to another the same way you would go from one node to another (like accessing the parent node, the preceding sibling node, attribute nodes etc). Strings are not nodes, hence once you have a string you can do pretty much everything with it without having to worry about whether you stylesheet is *guaranteed streamable* or not.

3. XSLT 3.0 streaming terminology

The following paragraphs explain a few essential terms in relation to streaming. For a more thorough coverage of streaming terminology, and the next section, initiating streaming, please refer to my previous paper [BRA14], where these subjects are explained in more detail and with more examples.

[1] The specification does, however, require that a processor, when operating in a streaming mode, must process the input data in constant memory, which implicitly requires the output stream to be flushed in time to meet that requirement.

[2] See section 14.8.6 of XPath Functions and Operators.

3.1. Guaranteed streamability

Streaming is all about *guaranteed streamability*[1]. Knowing that your stylesheet rules, when they apply to streaming, are guaranteed streamable is important, because it means it will be processed in a streaming way on every streaming processor, that is, on any processor that supports the streaming feature. It is possible that individual vendors have created ways to allow a broader group of constructs or expressions to be streamable, but that is out of the scope of this paper. Guaranteed streamability is well defined in the XSLT 3.0 specification, but the rules are complex.

This paper shows the application of a simplified set of rules to existing scenarios and how to turn stylesheets that are not *guaranteed streamable*[2] into stylesheets that are.

3.2. How to find out whether your processor supports streamability

If you are uncertain whether your processor supports streaming, or whether the license for your processor entitles you to using it, you can use the new streaming system property `xsl:supports-streaming`[3], which returns the value `yes` or `no` depending on whether the processor supports it.

You can use this property with `xsl:use-when` or with XSLT branch instructions like `xsl:choose` to force streamed processing of your input. A processor that does not support streaming is still required to process your input the traditional way, but if you explicitly want your stylesheet to fail when it is used with a non-streaming processor, you can use for instance the instruction `xsl:message` with `terminate="yes"` to break processing in such cases.

3.3. How to initiate streaming

There are essentially three ways to initiate streaming, or five, if you count using `fn:unparsed-text-lines` and vendor-specific methods as well, but these are outside the scope of this paper. A brief overview of the three ways to initiate streaming:

1. *Using xsl:mode:* with `xsl:mode`[4], the attribute `streamable="yes"` means that the mode, and all templates within that mode, will operate in a streaming way. That means that all templates in that mode must be guaranteed streamable. When you set `streamable="yes"` on the default unnamed mode, all templates in the default mode must be guaranteed streamable. To initiate streamed processing with a streamable mode, instruct your processor, i.e. by using command line parameters, to use that mode as an initial mode, in which case the processor *must* process the initial input document by using streaming.
2. *Using xsl:stream:* the most obvious way to initiate streaming is perhaps the new XSLT 3.0 instruction `xsl:stream`[5]. This instructions takes a `href` attribute value template and processes the document it finds using streamed processing. While using the initial mode option allows a creative developer to use something else then a document node as initial streamable item, the `xsl:stream` instruction can only return the document node in a streaming way, similar to the functions `fn:doc` and `fn:document`.

   Recently, as a resolution to public XSLT 3.0 Bug 25173, the working group has decided to add a function, `fn:streaming-document-available`, or `fn:streaming-available`[6]. This function will take an href and try to resolve it by trying to read the `Prolog` section of the XML document. How the precise semantics will work out is yet unclear, but it is important to realize that a streaming document is not stable, because it is not kept in memory as a whole. As a result, this function will, at the very most, give a *hint* as to whether a document is available or not. On a subsequent read, it may not be there any longer. This in contrast to `fn:doc-available`, which is guaranteed to return true if it can read the whole document[7].

[1] See section 19.10 in XSL Transformations 3.0.
[2] Technically, it is not a stylesheet that is *guaranteed streamable* or not, it is a construct that is. It depends whether that construct needs to be applied to a streaming node, which is dependent on whether the construct is used inside a template that is in a streaming mode, whether it is used in streamed merging, or whether it is used within, or called by the instruction `xsl:stream`.
[3] See section 20.3.4 in XSL Transformations 3.0.
[4] See section 6.6.1 in XSL Transformations 3.0, and section 6.6.3 on Streamable Templates.
[5] See section 18.1 in XSL Transformations 3.0.
[6] At the time of this writing, there was no consensus yet on the chosen name. Follow the linked bugreport or review the next public working draft to find out what the definitive name of this function will be.
[7] This is only partially true. At user option, a processor may offer a non-stable version of `fn:doc`, which allows the processor to optimize memory usage by not having to keep the full XDM tree of the XML document in memory. As a byproduct, in such scenarios it it not guaranteed that the same document node is returned upon a subsequent invocation of that function with the same argument.

3. *Using xsl:merge:* a less common way to initiate streaming is by using the `xsl:merge`[1] instruction. This instruction can process multiple sorted documents in order and merge them into a single output. For instance, if you have a document with current users and one with new users, and you want to merge them together, assuming they are sorted, you can use this instruction to do so. The merge instruction takes its sources through `xsl:merge-source` children, each of which can take a streamable document if the attribute `streamable` is set to `yes`. In-depth discussion of this instruction is out of scope for this paper[2], but see Section 5.10, "Sorting" for an example and its applicability with sorting.

4. Brief overview of the Ten Rules of Thumb of streaming

The following ten sections serve as a quick refreshment on the basics of streaming. As with the previous section, these *Ten Rules of Thumb* have been more thoroughly discussed, and with ample examples, in my previous paper [BRA14]. If you do not have prior experience with streaming in XSLT, I recommend you read the corresponding sections in that previous paper.

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

Each template rule that is in a streamable mode can contain a maximum of one downward expression[3] in all of its immediate children. This is the most important rule to remember, but do see Section 4.2, "Rule 2: each individual construct can have a maximum of one downward expression", where we are widening the concept a little.

Example:

```
<!-- the whole template has
     one downward expression -->
<xsl:template match="amount">
  <xsl:copy>
    <!-- the next line has
         one downward expression -->
    <xsl:apply-templates select="price"/>
  </xsl:copy>
<xsl:template>
```

4.1.1. What are downward expressions?

A downward expression is an expression for which the processor is required to read further into the currently open stream. This is in contrast to a *motionless* expression, which does not require the processor to move its reading head[4]. Such downward expressions are, for instance, a child-select expression, such as `child:section`, or a descendant-or-self expression, such as `head//section`. Not all downward selections are also guaranteed streamable. For instance, the expression `following-sibling::customer` is not guaranteed streamable when it operates on a streaming node, but is a downward expression. The term used for such expressions is: *free-ranging* expressions, which is a group of expressions that also includes non-downward expressions, such as `ancestor::head/section`, which is an expression that first moves up, and then down again.

More on how to determine what expressions are allowed in what constructs, is explained in the following sections.

[1] See section 15.2 in XSL Transformations 3.0.

[2] The current Public Working Draft [XSLT3] has some issues in regards to streamability of `xsl:merge`, making it impossible to do streaming, as I reported in [BRA14]. The most relevant bugs are XSLT 3.0 Bug 24343 and 25335. Both are meanwhile resolved.

[3] The specification talks about at most one *consuming* expression. In the body of the text of this paper, I typically prefer the term *downward expression*, though sometimes it seems more applicable to use *consuming*. Either term covers the same base.

[4] When I use the term *reading head*, it refers to a fictitious current position of the read-pointer of the processor in the opened streamed document. Whether or not a processor actually has such a read pointer, is an implementation detail. An optimizing processor might, for instance, do some look-ahead and caching, to improve up processing speed. Moving the reading head forward is literally the same as reading the next bytes, and thus nodes, from the input stream.

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

Let us extend the previous rule a little bit by saying that each construct has a maximum of one downward expression. In streaming, a *construct* is an instruction, an expression, a sub-expression (in `a | b` there are two subexpressions `a` and `b`) and declarations, if they are of influence to streamability. Typically, many constructs have a sequence constructor (such as `xsl:for-each`, `xsl:sequence`, `xsl:with-param`). This sequence constructor is at the heart of XSLT which allows us to nest instructions inside other instructions. A sequence constructor is itself a *construct* as seen in streaming analysis.

It is important to distinguish between focus-changing constructs, such as `xsl:for-each` and non-focus-changing constructs. When a construct changes focus, the expression changing the focus (usually the `select` attribute) can be a downward expression *and* the sequence constructor inside the container can have a downward expression.

Another focus-changing construct is the path operator. In an expression such as `a/b/c`, each of `a`, `b` and `c` is considered a construct[1], because each `/`-operator changes the focus. That is why it is allowed to write essentially *three* downward expressions in `a/b/c/`.

Conversely, the container of `xsl:if` does not change focus. As a result, the whole `xsl:if`-instruction is considered on construct, and either the `test`-attribute or the sequence constructor can contain a downward expression, but not both.

Filter expressions, as in `head/section[para]` do not change focus either, and since you already have a downward select in the other part of the expression, the `para` child-select expression is considered the second downward expression. The whole expression `head/section[para]` is therefor not streamable. See Section 4.9, “Rule 9: Use motionless filters” for more on filters.

4.3. Rule 3: Use motionless expressions where possible

A motionless expression, or construct, is one that doesn't require the processor to move from the current point in the input XML stream at all. Such expressions are expressions that do not operate on nodes, or expressions that request information from a node, such as its name, and can give that information without moving the reading head. Some examples of motionless expressions are:

- `fn:name()`, `fn:local-name` and `self::foo`. Also functions such as `fn:has-children`[2], `fn:exists` and `fn:in-scope-prefixes`.
- Climbing the parent, ancestor or ancestor-or-self axes (but not downward again after you climb). The reason this is allowed is that during streaming, the processor keeps a cache of all the ancestors of the current node.
- The attribute axis. Similar to the previous rule, attributes are considered part of the information of the current node. Combining the previous rule and this rule is possible, for instance, `author/ancestor::*/@company-name` is a guaranteed streamable expression.
- Atomic functions or functions that operate on atomics, as long as the current node is not an argument. The expression `fn:string(math:pow(3.14))` is motionless, but the expression fn:string(math:pow(input)) is not, the second has a child expression as argument and is, on itself, a downward expression and *not* motionless.
- Literal result elements or node creation instructions (unless of course, they contain an AVT[3] a TVT[4] or a sequence constructor that has a downward expression).
- Variable references are motionless[5] unless they occur inside `xsl:function` and are bound to a *streaming argument*[6]. Note that it *is* possible to create *copies* of nodes and bind those to variables, see Section 4.6, “Rule 6: Break out of streaming abundantly”.

[1] The specification differentiates between constructs and operands. In fact, both the `xsl:for-each` and the path expression has several operands, each of which follows certain rules. The instruction and path expression as a whole are considered a construct in the specification. This paper does not make that distinction, which in many cases makes analysis easier. In those cases where it is not applicable, I will explicitly mention it.

[2] Intuïtively, the function `fn:has-children` requires look-ahead. But since it only requires a very small look-ahead, it is considered streamable. Implementations just need to make sure they don't actually move the reading head.

[3] The term AVT means attribute value template, a way of writing an expression as a value of an attribute using `{expr}` syntax.

[4] A TVT, or text value template, is a new feature of XSLT 3.0 which allows you to intermix expressions with text nodes, see also section 5.7.2 in XSL Transformations 3.0.

[5] It is not allowed in streaming to bind a variable to a streaming node.

[6] This is a notable difference between my previous paper [BRA14] and this one. It is not possible anymore to bind grouping or merging variable using `bind-group`, instead, using `fn:current-group` and friends is legal, with certain restrictions, in streaming, see XSLT 3.0 Bug 24510. Conversely, it is now possible to write stylesheet functions that take at most one streaming argument (and argument that can be a reference to a node), see XSLT 3.0 Bug 25679.

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

The ancestor axis, together with the attributes and properties of the nodes, is the only axis that is kept in memory while performing streaming. As a result, you can consider the ancestor axis as a stack that is always available. However, while you can create a motionless expression like `parent::x/@name`, you are not allowed to move away again, because moving down would mean that you move the input read pointer in an arbitrary direction[1]. An example of such a *free-ranging* expression is `parent::x/address`, because that expression moves up, and then selects the child (moving down).

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

This rule was already touched upon in Section 4.3, “Rule 3: Use motionless expressions where possible”. It is very easy to master: you cannot store a reference to a streaming node, *ever*[2] . A reference to a node can be created within a `let` expression, an `xsl:variable` or an `xsl:param` instruction, or implicitly, as an argument to a stylesheet function[3] .

Example that is not guaranteed streamable:

```
<xsl:template match="log" mode="streaming">
  <!-- not allowed:
         binding a node to a variable -->
  <xsl:variable select="detail" name="dtl"/>
  <xsl:value-of select="$dtl"/>
</xsl:template>
```

Example that is guaranteed streamable[4]:

```
<xsl:template match="log" mode="streaming">
  <!-- allowed: creating a document
               node with a copy of a node -->
  <xsl:variable name="dtl">
    <xsl:copy-of select="detail"/>
  </xsl:variable>
  <xsl:value-of select="$dtl"/>
</xsl:template>
```

4.6. Rule 6: Break out of streaming abundantly

In Section 4.5, “Rule 5: You cannot store a reference to a node” you have seen an example that created a copy of a node and stored that inside a variable. This comes in handy, but such a copy does no maintain information from its ancestors, and creating a copy that does maintain that information is not obvious[5]. For cases where an inline expression should break out of streaming and create a copy of a node, maintaining all the information from its ancestors, XSLT 3.0 introduced the new function `fn:snapshot`. If the ancestors are not a requirement, you can use the lighter-weight funtion `fn:copy-of`.

Use these functions abundantly, possibly in conjunction with `xsl:copy-of`. They create an in-memory copy of whatever node you feed them, and on that copy, you can use any free-ranging expression that you like.

One caveat: usually, you will choose a streaming scenario for a reason, for instance because the input is too large to fit in memory. Do not use these functions on nodes that are potentially too large to fit in memory. Use them on nodes that you know are small enough to be processed one at a time in memory[6].

4.7. Rule 7: Understand streamable patterns

In essence: patterns must be motionless. Imagine any current node and its available information, like its ancestors, its attributes and its properties. If you consider a pattern as a function that returns *true* or *false*, and if that function can be applied without moving away from that current node, then it is a motionless pattern.

[1] This is called *free ranging* in the specification.

[2] A processors is allowed to extend the streamability rules. However, the term *guaranteed streamability*, which this paper discusses, is guaranteed across different streaming processors. Vendor extensions are out of scope for this paper.

[3] Since the resolution of XSLT 3.0 Bug 25679 on Streamable stylesheet functions, it is now possible for a stylesheet function parameter to be bound to a node. As a consequence, it is possible for a variable reference, if bound to such a parameter, to be bound to a node. The resolution of this bug is not public, but the essence is important for writing library packages catered for streaming, which would otherwise not be possible.

[4] The example here uses the fact that the sequence constructor, by default, creates a new document node. In addition, the `xsl:copy-of` instruction creates a copy of the node too, disconnecting it from the streaming node and consuming the whole node at once.

[5] For an example of how complex this is to do by hand, see the 75 lines long formal function that defines `fn:snapshot` in section 18.4 of XSL Transformation 3.0

[6] A processor is not required to dismiss memory after one of these functions are used, but it is likely that implementations will dismiss off memory of copies that go out of scope, otherwise, these functions would have little merit in long-running streaming scenarios.

A pattern behaves different from expressions. Take an select expression such as `book/chapter`. As an expression, this is considered a downward expression, because it *selects* the nodes below the current node, so it must move the reading head. However, patterns are evaluated on every node when they are encountered. When the processor encounters a node with the name `section`, it knows instantly, without looking ahead or backwards that the pattern does not match the right-hand part `chapter` of the expression: the name does not match. If it finds a node with the name `chapter`, it can check the parent without moving the reading head whether it matches `book`. Conclusion: `book/chapter` is a *motionless pattern*[1] and therefore streamable.

All patterns that do not start with a variable reference or a function call, and that do not have a predicate, are streamable. Examples are `html//div`, `a//b//c//d`, `child:num`, `author/@name` and `author/@name`[2].

If your pattern has a predicate, that predicate must be motionless, similar to Section 4.9, "Rule 9: Use motionless filters", but must not be a numeric predicate. In nested predicates, however, you can use a numeric value or `fn:position`, but they still must be motionless. Valid examples are `author[@name]`, `author[ancestor::guild]` and `author[ancestor::*[3][self::publisher]]`, the latter having a numeric predicate, which is allowed because the predicate is nested.

4.8. Rule 8: Use atomic types to ground the result of templates

Any template rule or named template must be *grounded*[3], which means, it is not allowed to return streaming nodes. This rule is very similar Section 4.5, "Rule 5: You cannot store a reference to a node". Since using `xsl:sequence` inside bodies of `xsl:template` is very common, and since `xsl:sequence` return a reference of a node, you should be careful about using them.

But there is an easier way out. Instead of trying to keep track on whether or not your template returns nodes, it is much simpler to correctly set an atomizing type[4] for your template. When a type is set for your template, the processor must atomize the result anyway, so it *knows* statically that the template will never return any nodes. This prevents you from worrying about whether or not you are returning nodes or not. The principle of *taking the result type into account* also applies to other constructs, such as the parameter and return types of stylesheet functions and templates, or the declarations of variables.

If you want to return more complex content than an atomic type, use `xsl:copy` and similar constructs that create copies of nodes, as opposed to `xsl:sequence`, which returns references to nodes.

4.9. Rule 9: Use motionless filters

In earlier chapters we have already seen what a motionless expression is. In filter expressions, that is, in a predicate, you should always only use motionless expressions, or the result will not be streamable. Some examples:

```
price/text()[contains(., 'father')]
child::node()/copy-of(.)[contains(., 'father')]
helptext[@version = '3.14']
.//comment()[contains(., 'father')]
p[parent::div][ancestor::p][@xml:lang]
```

All the above expressions use motionless filters, but they are motionless for different reasons.

The first is motionless because `text()`-nodes considered leave-nodes (they cannot possibly have children). A filter expression on a guaranteed childless node may consume that whole node, here done by the context-item expression `.`.

The second is motionless because we first created a copy of the whole node using `copy-of`, after that, anything we do on that node, can be any regular XPath expression, see also Section 4.6, "Rule 6: Break out of streaming abundantly".

The third tests the value of an attribute, attributes are always available as part of the stack the processor keeps of each node: an attribute is a property of a node and henceforth it is motionless.

[1] See section on Classifying Constructs in XSL Transformation 3.0.
[2] Recall that a pattern is not the same as an expression. For instance, you cannot write path expression using the following-sibling or parent axis. This was true in XSLT 1.0 and 2.0 and remains true in XSLT 3.0.
[3] *Grounded* means that it cannot return streaming nodes.
[4] An atomizing type is any type that is not a node type, or cannot be a node type. Examples are `xs:string`, `xs:integer`, `xs:double*`, but not `element()`, `attribute()` or `item()`.

The fourth is a filter on a comment. Similarly to text-nodes, comments can never have children, so the expression is allowed to consume the whole node and still be considered motionless[1].

The last example combines three motionless filters and searches for illegally nested `p` elements in an HTML document, that also have the `xml:lang`-attribute set. The parent axis, the ancestor axis and the attribute axis are all part of the available properties that are maintained by the processor as a stack during processing, therefore this expression is motionless too.

4.10. Rule 10: Master xsl:fork

There is an alternative for scenarios where you would otherwise require multiple downward selects in one construct. Use `xsl:fork`[2], which explicitly tells the processor that from this point on, it should start multiple threads for processing the input stream in parallel. In other words, the stream reader will get multiple read pointers that all have their own starting point, the current node. This process, called *forking* allows multiple downward selections in a single instruction, but is very strictly defined.

The rules for using `xsl:fork` are briefly as follows:

- It can only contain `xsl:sequence` children
- Each `xsl:sequence` can have at most one downward select expression, as explained in Section 4.2, “Rule 2: each individual construct can have a maximum of one downward expression”.
- Each `xsl:sequence` must be grounded, as in Section 4.8, “Rule 8: Use atomic types to ground the result of templates”, it cannot return nodes, but this time you cannot use the `as`-attribute to the rescue, so type-determined usage, as also discussed in the same *Rule Eight*, cannot be applied[3].

Once you have these rules in place, you are allowed multiple selections in a single construct, you can *fork* the processing. In fact, that is exactly what you tell the processor: from hear on in, I want multiple reading heads on the same stream.

Example of splitting logfile entries between warnings and errors[4]:

```
<xsl:fork>
 <xsl:sequence>
   <errors>
     <!-- one downward select -->
     <xsl:copy-of select="entry[@type eq 'err']"/>
   </errors>
 </xsl:sequence>
 <xsl:sequence>
   <warnings>
     <!-- and another one, but still legal! -->
     <xsl:copy-of select="entry[@type eq 'warn']"/>
   </warnings>
 </xsl:sequence>
</xsl:fork>
```

5. Streaming Design Patterns

In the following sections, I will show programming design patterns that are common in traditional, non-streaming scenarios, but that are themselves roaming and free-ranging expressions or constructs. As a result, they cannot be used in a streaming scenario by copying them one-on-one.

The streaming design patterns apply both to converting existing stylesheets, and making them streamable, as to new stylesheets that have to be streamable from the beginning.

Each pattern follows the same format:

- *Intent:* a brief summary of the patterns intent.
- *Level:* an indication for the level of experience required to understand an implement the pattern[5]. Under the *Level* section, I will also mention the level of streamability the solution has after applying the streaming pattern, in order, from *full* to *none*:

[1] In fact, this is precisely how the specification defines it: consuming a childless node is considered a motionless operation. This may seem like a constradiction in terms, but helps a lot with keeping the analyzing terms to a minimum. The official rule is, however, not very easy to read and understand: *If U is absorption and T is a childless node kind (text(), attribute(), comment(), processing-instruction, or namespace-node()), then U' is inspection*, see Section 19.8, General Streamability Rules in XSL Transformation 3.0.

[2] See also section 16.1 in XSL Transformations 3.0

[3] In practice this means that you must wrap the sequence constructor of each `xsl:sequence` in `xsl:copy` of `xsl:value-of`or likewise constructs that return copies, not references to nodes.

[4] Example was directly taken from [BRA14], page 21, where you can find more information on using `xsl:fork`.

[5] The level indicated is solely based on my experience within the XSLT 3.0 Working Group, where certain subjects appeared to be much harder to be understood by the group members, while others were relatively easy to grasp.

  - *Full:* means the streaming pattern is fully streamable, without buffering or copying nodes.
  - *Implicitly windowed:* technically the same a *full*, but applies to constructs such as `xsl:try`[1] and `xsl:merge` where the current merge group is implicitly copied, and thus grounded[2].
  - *Modified windowed:* same as *windowed* below, but first a limited, modified copy of the nodes is created to prevent taking up too much memory.
  - *Windowed:* the streaming pattern uses windowed streaming, meaning that a copy of a node or nodes is created in memory to do further processing on without the restrictions of streaming.
  - *Multiple pass:* the streaming pattern uses multiple passes on the input document; this ensures limited use of memory, but may (dramatically) increase processing time.
  - *None:* no streaming solution exists.
- *Motivation:* explains why this pattern does not work using the traditional, non-streamable approach.
- *Applicabilty:* a summary of typical scenarios this pattern applies to.
- *Consequences:* an enumeration of possible drawbacks of the pattern.
- *Implementation:* step-by-step guide to implement this pattern on existing code.
- *Example:* applies the design pattern and applies it to the example from the *Motivation* section.

5.1. Filter expression depends on children

5.1.1. Intent

You have a select expression with a predicate that depends on data in the children and want to make it streamable.

5.1.2. Level

Intermediate: uses techniques available in XSLT 2.0. Streamability level: modified windowed.

5.1.3. Motivation

Suppose your data structure looks something like the following, where the elements `largedata` contain potentially large sections of data that you do not want copied, for instance base64 formatted images or other data related to the person:

```
<people>
    <person>
       <largedata>
       <name>...
       <largedata>
       <address>
          <largedata>
          <street> ...
          <number> ...
          <city> ...
          <state>MA</state>
          <largedata>
       </address>
       <largedata>
    </person>
    <person>
       ....
</people>
```

If your requirement is to copy certain fields from `person`, based on whether they live in a certain state, you may have originally used a coding pattern like the following:

```
<xsl:template match="people">
  <xsl:apply-templates
    select="person[address[state = 'MA']]"/>
</xsl:template>
```

In a streaming scenario, this will not be guaranteed streamable, because inside the `select` attribute, the expression uses two downward selects in two filter expressions, and we know from Section 4.9, “Rule 9: Use motionless filters” that that is not allowed. The select expression would not even have worked if you had only one downward expression inside the predicate.

A quick and easy way to fix this example is the following, applying Section 4.6, “Rule 6: Break out of streaming abundantly”:

```
<xsl:template match="people" mode="streaming">
  <xsl:apply-templates
    select="copy-of(person)[
              address[state = 'MA']]"/>
</xsl:template>
```

This works, but what if the data of one `person` is too large (for instance, the record contains binhex image or video data that you would like to strip), or the memory constraints too streneous (as on mobile or embedded devices), such that in your case `fn:copy-of` selects too much data and blows up the processing?

[1] The instruction `xsl:try` deserves special attention. It does not create copies of streamed input, but it buffers (creates a copy) of streamed output, because in the event of a failure, the processor is required to rollback to the point before the `xsl:try` instruction.

[2] I would like to liberally quote Michael Kay here as I heard him explaining this concept: “in the vast majority of cases, the implicit copy will be small, only a few pathological cases may require a large copy, in which case it is up to the stylesheet author to come up with a better solution”. This change is not currently in the public spec, however see XSLT 3.0 Bug 25335.

In such cases, use this pattern to use a guaranteed streamable approach to create a partial copy of the offending element instead.

5.1.4. Applicability

Applies to

This pattern applies to almost any place where you can use an expression:

1. Select expressions as used in the likes of `xsl:copy-of`, `xsl:copy`[1], `xsl:apply-templates`, `xsl:for-each`, `xsl:for-each-group` and `xsl:message`;
2. Select expressions in `xsl:with-param` and `xsl:variable`, which works, because the result of the pattern below is guaranteed grounded.;
3. Select expressions that are atomized or require a boolean, like in `xsl:analyze-string`, `xsl:if`, `xsl:when`, `xsl:comment` and `xsl:processing-instruction`;
4. Expressions used with atomizing arguments of any internal or user function, such as where the argument is `xs:string` or `xs:integer*`.

Does not apply to

This pattern does not apply to:

1. Matching patterns, see Section 5.2, "Patterns with non-motionless predicates".

5.1.5. Consequences

Applying this pattern has the following consequences:

1. Extra code for creating a partial copy of the node may impede maintainability.
2. With more complex expressions, it can get cumbersome to split it up, in which case you should look whether you can use windowed streaming, using `fn:copy-of` or `fn:snapshot`. But that may not be possible if the selected nodes are too large to begin with.

5.1.6. Implementation

The essence of this pattern is to create a limited copy of the node before you apply the predicate, and then apply the predicate on this copy, which is allowed in streaming. The copy will only contain the leaf elements we are interested in, not the large data sections such as `largedata` from our example.

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode and set the attribute `on-no-match="shallow-copy`[2].
2. Remove the offending filter expression.
3. Create a new matching delete-template that matches the large nodes we want to skip.
4. Add a template that matches the offending element.
5. In it, add a variable to create a partial copy of the offending element.
6. Process this variable with your original predicate and templates. This works, because variable references are grounded by default and you can use free-ranging expressions on them[3].

[1] Since XSLT 3.0, it is possible to have a `select` attribute on `xsl:copy`, see section 11.9.1 on shallow copying in XSL Transformation 3.0.

[2] The on-no-match mode `shallow-copy` will copy any non-matching nodes, which allows us to remove only the nodes we are not interested in, without changing the whole stylesheet, this attribute, as well as the `xsl:mode` declaration, are new features of XSLT 3.0.

[3] Swtiching modes from a streaming mode to a non-streaming mode is allowed, as long as the select-expression does not return nodes from the streamed document, which is why this pattern creates a copy of a subset of the nodes. Alternatively, you can stay inside the new streaming mode, but this restricts the bodies of the templates.

Example

If we apply these six steps to our original example, we end up with the following code:

```
<!-- (1) add a streamable mode -->
<xsl:mode name="streaming" streamable="yes"
          on-no-match="shallow-copy"/>

<!-- (1) use the streamable mode -->
<xsl:template match="people" mode="streaming">
  <!-- (2) remove the filter-expression -->
  <xsl:apply-templates select="person"
                       mode="#current"/>
</xsl:template>

<!-- (4) add a template matching "person" -->
<xsl:template match="person" mode="streaming">

  <!-- (5) introduce a variable
           to copy partially -->
  <xsl:variable name="person-partial">
    <xsl:copy>
      <xsl:apply-templates select="@*"
                           mode="#current"/>
      <xsl:apply-templates select="node()"
                           mode="#current"/>
    <xsl:copy>
  </xsl:variable>
  <!-- (6) continue in the non-streaming
           mode on the elementcopy -->

  <xsl:apply-templates select="$person-partial/
    person[address[state = 'MA']]"/>
</xsl:template>

<!-- (3) create a delete-template
         for the large nodes -->
<xsl:template
  match="node()[ancestor-or-self::largedata]"
  mode="streaming">
  <xsl:apply-templates mode="#current"/>
</xsl:template>

<!-- (3) create a matching pattern
         for the inclusive elements -->
<xsl:template
  match="address/*[not(self::largedata)]"
  mode="streaming">
  <xsl:copy-of select="."/>
</xsl:template>
```

In the above code, you can see that we specify what parts of the streaming input data we want to delete[1], so that the copy of the nodes is as small as possible, i.e. without any of the large nodes. The result, a handful of nodes with string data as content, is copied into a variable. In turn, the contents of this variable, which is now *grounded*, is copied to the final result tree, using the same predicate we used previously.

Note the special variant of the identity template. Instead of placing `node | @*` in one select-statement, it is split into two. The reason behind this is that a *climbing*[2] expression and a *striding*[3] expression cannot be combined in a single `union` expression. However, a climbing expression that only selects leaf nodes[4] is allowed on its own in a select-satement, as is a striding expression.

The benefits of using this pattern is that you can leave much of your original code intact. We merely removed the nodes that made streaming impossible, and then continued processing on a small copy of the nodes.
After we have applied our changes, the total amount of memory required for this whole operation is much less than in the first solution, where we copied the entire `<person>` element as a convenient quick solution. While there are methods you can use that may use even less memory, for instance with accumulators and maps, but these approaches are far harder to implement and in this particular scenario may only gain a couple of bytes for you during processing.

5.2. Patterns with non-motionless predicates

5.2.1. Intent

You have a matching pattern that depends on its children and want to make it streamable.

5.2.2. Level

Easy: uses common, well-known XSLT 2.0 techniques. Streamability level: windowed.

[1] There are many ways of writing an appropriate delete-template. I chose for `ancestor-or-self`, but you may require a more fine-grained approach in your situation. Remember that any nodes not specifically speficied in a matching pattern are copied as a result of `on-no-match="shallow-copy`.

[2] A climbing expression is an expression that *climbs up* the tree, as we have seen in Section 4.4, “Rule 4: You can move up the tree, but never down again”. Attributes are considered *climbing*.

[3] A striding expression is an expression on the child axis or a combination of child axes, where overlapping nodes will not occur.

[4] As seen before, a leaf node is a node without children. Attributes and text-nodes are leaf nodes, but a climbing expression like `ancestor::title` is not, because `title` can have children.

5.2.3. Motivation

Suppose you want to select all `para` elements that contain an `emphasis` element and you want to remove all other `para` elements. In a non-streaming scenario, your stylesheet could look something like this:

```
<xsl:template match="para[descendant::emphasis]">
  <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="para"/>
```

This pattern is not streamable, because it requires the processor to look down inside all possible children of the `para` element and after that, it should go back and copy everything from beginning to end.
In fact, as explained in Section 4.7, “Rule 7: Understand streamable patterns”, a predicate in a pattern must be motionless. Because you are not allowed to use *windowed template matching* using `fn:snapshot` or `fn:copy`[1], this pattern's solution takes an old-school imperative approach to solve the non-motionless predicate issue.

5.2.4. Applicability

Applies to

This pattern applies, among others, to the following scenarios:

1. Match patterns with predicates based on properties of descendant nodes.

Does not apply to

This pattern does not apply to the following scenarios:

1. Match patterns with predicates based on the following-sibling or preceding-sbiling axes;
2. Match patterns with predicates based on the following or preceding axes.

5.2.5. Consequences

Applying this pattern has the following consequences:

1. The decision-tree must imperatively be written by hand, instead of letting the processor define the logic for you, which is the normal operation mode with template based matching.
2. Tricky to get right when multiple template rules exist that process the same element. In such cases, it is better to use windowed streaming, or, if that is not an option, forking.

5.2.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode.
2. Remove the delete-template and the predicate in the match-pattern.
3. Create a copy of the node.
4. Rewrite the decision logic of the predicate inside the template, or apply templates on the copy.

Example

If we apply these three refactoring rules to our original example, we end up with the following code:

```
<!-- (1) change the mode to streamable -->
<xsl:mode streamable="yes"/>

<!-- (2) remove the predicate -->
<xsl:template match="para" mode="streaming">
  <!-- (3) create a copy of the node -->
  <xsl:variable name="copy"
    select="copy-of(.)"/>
  <!-- (4) rewrite the decision logic -->
  <xsl:copy-of
    select="$copy[descendant::emphasis]"/>
</xsl:template>

<!-- (2) remove the delete-template -->
```

After applying this pattern, we have inlined the decision-logic, resulting in a template body with one downward select to create a copy of the current node[2], which is streamable, because any subesequent processing on this copy will be grounded, and therefor allowed during streaming.

This streaming pattern applies to any non-motionless predicate that, after rewriting it as (something similar as) above, ends up with a maximum of one downward select per template. If you end up with multiple downward selects after rewriting, then also apply the following pattern, *Instructions with multiple downward selects*.

[1] Windowed template matching does not exist (yet), but it borrows the term from *windowed streaming*, which creates a copy of a small subset of nodes. Windowed template matching could look like `para[copy-of(.)/descendant::emphasis]`, but this is not legal in streaming, and outside of streaming the `fn:copy-of` has no effect.
[2] If the copy will be too large, use the first pattern to preprocess it to do a modified windowed streaming approach.

5.3. Instructions with multiple downward selects

5.3.1. Intent

You have an instruction with multiple downward selects in document order and want to make it streamable.

5.3.2. Level

Easy: follows a commong refactoring pattern from the XSLT 2.0 community. Streamability level: full.

5.3.3. Motivation

Consider the following example:

```
<xsl:template match="address">
  <span><xsl:value-of select="street"/></span>
  <span><xsl:value-of select="number"/></span>
  <xsl:apply-templates select="state | country"/>
</xsl:template>
```

This template is not streamable, because it violates Section 4.1, "Rule 1: each template rule can have a maximum of one downward expression", it has multiple downward selects, here with the select expressions `street`, `number` and `state | country`.

5.3.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. The body of any instruction that has multiple downward selects in document order;
2. The body of a streamable stylesheet function having multiple downward selects in document order;
3. The body of a template declaration having multiple downward selects in document order.

Does not apply to

This pattern does not apply to:

1. Function-calls that contain multiple downward selects;
2. Single expressions with multiple downward selects;
3. Instructions with multiple downward selects that are not in document order, see Section 5.4, "Instructions with multiple downward selects out of document order".

5.3.5. Consequences

Applying this pattern has the following consequences:

1. After rewriting the body, its coherency may be harder to understand and thus harder to maintain.

5.3.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode with `on-no-match="deep-skip"`[1];
2. Replace the body of your instruction with a single `xsl:apply-templates`;
3. Add matching templates for the individual elements.

Example

Applying these three refactoring rules to our original example, we end up with the following code:

```
<!-- (1) change the mode to streamable -->
<xsl:mode streamable="yes"
          on-no-match="deep-skip"/>

<!-- (1) change the mode to streamable -->
<xsl:template match="address" mode="streaming">
  <!-- (2) replace the body with
           a single apply-templates -->
  <xsl:apply-templates select="*" mode="#current"/>
</xsl:template>

<!-- (3) add matching templates
         for each element -->
<xsl:template match="street | number"
              mode="streamable">
  <span><xsl:value-of select="."/></span>
</xsl:template>
<xsl:template match="state | country"
              mode="streamable">
  <!-- details left out -->
</xsl:template>
```

This pattern is perhaps the most common pattern with streaming: take any given instruction with too many downward selects and split it up in smaller pieces to make it streamable. Most people will be familiar with this pattern, because it does not introduce any new concepts, in fact, it is a common refactoring method in existing stylesheets.

[1] `deep-skip` skips non-matching nodes without processing their children.

5.4. Instructions with multiple downward selects out of document order

5.4.1. Intent

You have an instruction with multiple downward selects that do not follow the input document order and want to make it streamable.

5.4.2. Level

Advanced: introduces a new concept, forking. Streamability level: full.

5.4.3. Motivation

Consider the following example:

```
<xsl:template match="address">
  <!-- number first -->
  <span><xsl:value-of select="number"/></span>
  <span><xsl:value-of select="street"/></span>
  <xsl:apply-templates select="state | country"/>
</xsl:template>
```

Similar to Section 5.3, “Instructions with multiple downward selects”, this template is not streamable, because it violates Section 4.1, “Rule 1: each template rule can have a maximum of one downward expression”, it has multiple downward selects, again with the select expressions `street`, `number` and `state | country`. Note, however, an important difference with the previous design pattern: in this case, the order of the selected elements does not match the document order of the input document.

5.4.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. The body of any instruction that has multiple downward selects in any order;
2. The body of a streamable stylesheet function having multiple downward selects in any order;
3. The body of a template declaration having multiple downward selects in any order.

Does not apply to

This pattern does not apply to:

1. Function-calls that contain multiple downward selects;
2. Single expressions with multiple downward selects;

5.4.5. Consequences

Applying this pattern has the following consequences:

1. The instruction `xsl:fork` has no effect on the result, it can be hard for programmers to understand why it is introduced and the temptation to remove it at a later stage, because "it does nothing special" can be big.
2. Code quickly gets messy, because nesting gets two levels deeper with each fork.

5.4.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode;
2. Wrap each downward select in an `xsl:fork` instruction.

Example

Applying these two refactoring rules to our original example, we end up with the following code:

```
<!-- (1) change the mode to streamable -->
<xsl:mode streamable="yes"/>

<!-- (1) change the mode to streamable -->
<xsl:template match="address" mode="streamable">
  <!-- (2) wrap in xsl:fork -->
  <xsl:fork>
    <xsl:sequence>
      <span><xsl:value-of select="number"/></span>
    </xsl:sequence>
    <xsl:sequence>
      <span><xsl:value-of select="street"/></span>
    </xsl:sequence>
    <xsl:sequence>
      <xsl:apply-templates
        select="state | country"/>
    </xsl:sequence>
  </xsl:fork>
</xsl:template>
```

As explained in Section 4.10, “Rule 10: Master xsl:fork”, forking is a technique that has no semantic meaning, it merely tells the processor to process the streaming input from this moment on using multiple reading heads. Introducing `xsl:fork` in code allows the programmer to write a sequence of `xsl:sequence` elements, of which the body, or select attribute, can each contain a single downward select expression.

This pattern opens up a lot of possibilities. It is a common scenario to change the order in which elements are output. Using `xsl:fork` makes this easier to achieve using streaming[1].

5.5. Expressions with the preceding- or following-sibling axes

5.5.1. Intent

You have an expression that involves iterating over the preceding-sibling or following-sibling nodes[2].

5.5.2. Level

Advanced, involves streamed grouping. Streamability level: full.

5.5.3. Motivation

In flat-to-hieararchical scenarios it is still quite common to use the following-sibling axis. Consider the following input document:

```
<books>
    <book type="classic"/>
    <title>Don Quixote</title>
    <author>Micuel De Cervantes</author>
    <book type="classic"/>
    <title>Pilgrim's Progress</title>
    <author>John Bunyan</author>
    <book type="classic"/>
    <title>Robinson Crusoe</title>
    <author>Daniel Defoe</author>
</books>
```

and we would like to turn this into a nested structure, where `title` and `author` are part of the `book` element. One way of doing that is:

```
<xsl:template match="/books">
  <xsl:copy>
    <xsl:apply-templates select="book"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="book">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates
      select="following-sibling::title[1]"/>
    <xsl:apply-templates
      select="following-sibling::author[1]"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="title | author">
  <xsl:copy-of select="."/>
</xsl:template>
```

In practice, usage of the following sibling and preceding sibling axis has been getting less and less attention since the dawn of XSLT 2.0, which introduced grouping[3]. One of the reasons we don't see much of the sibling axis is that sibling recursion is quite hard to get right in XSLT 2.0 and up[4] and because most coding patterns are just easier to accomplish and understand with `xsl:for-each-group` with `group-adjacent` or `group-starting-with`, as is explained in [MKAY08], page 116.

The output of the above template would be the following:

```
<books>
   <book type="classic">
      <title>Don Quixote</title>
      <author>Micuel De Cervantes</author>
   </book>
   <book type="classic">
      <title>Pilgrim's Progress</title>
      <author>John Bunyan</author>
   </book>
   <book type="classic">
      <title>Robinson Crusoe</title>
      <author>Daniel Defoe</author>
   </book>
</books>
```

[1] Other ways of achieving the same result are not so trivial, unless you fallback to creating a copy of the node and process that copy using non-streaming templates, as in the first pattern.

[2] The axes following, preceding, following-sibling and preceding-sibling are part of the group of *forbidden streaming paths*, they cannot operate on a streaming node, if they do, the streamability analysis will deem them free-ranging. They are allowed, however, on grounded nodes, i.e., nodes that do not belong to a streamed document.

[3] It has not entirely disappeared, a recent thread on StackOverflow discusses this coding pattern, see XPath to select contiguous elements.

[4] In XSLT 1.0, if an axis selected multiple nodes, only the first one was actually selected. In XSLT 2.0, by default, *all nodes on an axis* are selected in path expressions. As a result, sibling recursion in XSLT 1.0, which typically relied on this feature, was not compatible and had to be rewritten for XSLT 2.0.

5.5.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Expressions that use the preceding-sibling or the following-sibling axes.
2. Can sometimes also be applied to patterns that use these axes.

Does not apply to

This pattern does not apply to:

1. Expressions with the following and preceding axes.

5.5.5. Consequences

Applying this pattern has the following consequences:

1. It makes your code more readable and easier to maintain[1].
2. Streamed grouping can be hard to grasp and to get right, which in more complex scenario may require a significant endeavor.

5.5.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streaming mode.
2. Replace the sibling recursion pattern with `xsl:for-each-group`[2].
3. If needed, adjust the resulting code using previous patterns, to make it streamable.

Example

Applying these three refactoring rules to our original example, we end up with the following code:

```
<!-- (1) introduce a streamable mode -->
<xsl:mode streamable="yes"/>

<xsl:template match="/books">
  <xsl:copy>
    <!-- (2) rewrite the sibling recursion -->
    <xsl:for-each-group select="*"
                        group-starting-with="book">
      <xsl:copy>
        <!-- (3) streamable body -->
        <xsl:copy-of select="@*"/>
        <xsl:copy-of select="current-group()[
                       position() > 1]"/>
      </xsl:copy>
    </xsl:for-each-group>
  </xsl:copy>
</xsl:template>
```

This streaming pattern follows a typical XSLT 3.0 scenario for grouping. By applying grouping we can get rid of the `following-sibling` axis steps. In this case, it sufficed to use `group-starting-with`, which takes a motionless pattern as seen from Section 4.7, "Rule 7: Understand streamable patterns". The `select`-statement is obvious and allowed in streaming (it selects children).

Let us have a look at the individual parts. The nesting applied here follows Section 4.2, "Rule 2: each individual construct can have a maximum of one downward expression", each construct in it has a maximum of one downward expression:

- The `xsl:copy` instruction is motionless if it does not have a `select` statement, however we still need to look at its sequence constructor, as per *Rule Two*.
- The `xsl:for-each-group` instruction is consuming, it contains one downward select. However, as we have learned in *Rule Two*, it is also a focus-changing construct. This means that the sequence constructor inside it can have yet another downward select expression. Because we already established that the pattern in `group-starting-with` is motionless, this does not count towards the limit of one, we can have as many motionless expressions as we want.

[1] That is a first! Usually, refactoring code to fit a streaming scenario results in less readable code, but this pattern actually makes it more readable and maintainable. Perhaps you should have applied this pattern already, even before you started streaming.

[2] This is not always trivial to get right, but many sibling recursion patterns can be rewritten with `group-starting-with` or with `group-adjacent`.

- The second `xsl:copy` instruction is again motionless on itself, but we need to check its sequence constructor.
- The first `xsl:copy-of` instruction is motionless, remember that visiting the attribute axis is always allowed, see Section 4.3, “Rule 3: Use motionless expressions where possible”.
- The second `xsl:copy-of` instruction is consuming. It consumes the `fn:current-group` and selects all elements except the first. In most cases, using or referencing the current group will count as a downward expression, as such it is not legal to have more than one usage of the expression, nor can you use the expression in an inner loop or loop expression[1].
- Note that, similar to the explanation in the first streaming pattern, it is not possible to combine these two `xsl:copy` instructions into one with a union expression[2].
- Note that using the `fn:position` function is allowed and motionless in any expression. It only has special meaning in patterns, where it is not allowed in a top-level predicate.

5.6. Expressions using the preceding axis

5.6.1. Intent

You have an expression that uses the preceding axis and want to make it streamable.

5.6.2. Level

Advanced: uses accumulators[3][4]. Steamability level: full.

5.6.3. Motivation

Suppose you want to keep a tracking word count per paragraph that shows all preceding words. You could code that as follows (though on large documents this will be highly inefficient):

```
<xsl:template match="text">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="para" use-when="false()">
  <xsl:copy>
    <xsl:attribute
      name="words-sofar"
      select="count(
        preceding::text()/tokenize(., ' '))"/>
    <xsl:copy-of select="text()"/>
  </xsl:copy>
</xsl:template>
```

The code above is so trivial that I will leave out the flat input XML and output XML. It simply counts all the words up until the current paragraph and adds this count to the `para` element as an attribute. To make this streamable, we will have to use accumulators[5].

5.6.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Path expressions using the following[6] or preceding axes.
2. Path expressions using the sibling axes[7].
3. Any free-ranging expression that cannot be made streamable using any of the other streaming patterns.
4. Any place where you need to keep (the equivalent of) a running total[8].

[1] The public Last Call WD in Section 19.8.8.5 specifically disallowed `fn:current-group` in streaming. After several bug reports and extensive research to assess whether allowing this function in streaming proved doable, a proposal by Michael Kay was accepted through XSLT 3.0 Bug 24510. See also XSLT 3.0 Bug 23391, 24342, 24317 (talks about a limitaion in grouped streaming), 24509, 24556, 24455. The resolution is publicly available as an attachment to Bug 24510, comment 9, which is a good starting point.

[2] Which is only partially true, you could do it by creating an inline copy of the attributes, such as `<xsl:copy-of select="copy-of(@*) | current-group()[position() > 2]" />`.

[3] If possible, it is *much* easier to solve this pattern using windowed streaming, by creating a copy of the parent node, which allows you to use any expression. However, that only works if the *distance* between the preceding node and its possible usages is small and fits in a windowed copy.

[4] Sometimes it is possible to rewrite such axes in terms of child axes, tunneled parameters or other ways, that would be streamable. If such a solution exists, it should take preference over using accumulators.

[5] This coding problem can also be solved using tunneled parameters, and in fact, it is believed that most accumulator-scenario can be expressed with tunneled parameters, however, it can get very unwieldy to keep track of tunneled parameters across all templates in a real-world scenario. Accumulators are a cleaner way to do this.

[6] It applies only to the following axis if the stylesheet can be rewritten using the reverse of that axis, because accumulator *cannot* look forward, they can only remember "what has been processed so far".

[7] Prefer the previous streaming pattern, where a sibling recursion is rewritten using streamed grouping.

[8] For instance, multi-level section numbering is a typical scenario for accumulators. An example, including explanation on how it works with accumulators, is in Section 18.2.8, Examples of Accumulators, in the XSL Transformations 3.0 Specification.

Does not apply to

This pattern does not apply to:

1. Scenarios where information ahead is required.
2. Scenarios that can be rewritten without using accumulators (this has preference).
3. Scenarios that cannot be expressed as a sequence of motionless expressions upon processed nodes[1].

5.6.5. Consequences

Applying this pattern has the following consequences:

1. May dramatically increase complexity by having to use accumulators.
2. Need to keep track of visiting of passed nodes, which can be hard to get right[2].

5.6.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode;
2. Create a streamable accumulator.
3. Add the appropriate accumulator rules to add properties to visited nodes.
4. Change the offending expression to use the accumulator expression instead.

Example

Applying these four refactoring rules to our original example, we end up with the following code:

```
<!-- (1) introduce a streamable mode -->
<xsl:mode streamable="yes"/>

<!-- (2) create a streamable accumulator -->
<xsl:accumulator name="count-words"
                 as="xs:integer"
                 initial-value="0"
                 streamable="yes">

  <!-- (3) add appropriate
           rules to decorate nodes -->
  <xsl:accumulator-rule
    match="text()[parent::para]"
    phase="start"
    new-value="$value + count(./tokenize(.,' '))"/>

</xsl:accumulator>

<xsl:template match="text">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="para">
  <xsl:copy>
    <!-- (4) adjust the expression
             to use the accumulator -->
    <xsl:attribute
      name="words-sofar"
      select="accumulator-before('count-words')"/>

    <xsl:copy-of select="text()"/>
  </xsl:copy>
</xsl:template>
```

Before we explain the code, let us first look at what accumulators are and how they can be applied:

An accumulator is best seen as a calculated property, or decoration, or function of a node: it can "attach" a value to a node that is made available through the `fn:accumulator-before` and `fn:accumulator-after` functions[3]. Accumulators are processed for *each* node that is processed in a streamable way[4], regardless of their source[5]. Streamable accumulators must be motionless and they cannot return streamed nodes.

[1] Accumulators can only operate on nodes with a motionless expression.
[2] Using maps makes this task a tad easier though, maps are a new type introduced in XSLT 3.0.
[3] For more information on accumulators and examples, see Section 18.2 in XSL Transformations 3.0.
[4] Provided its attribute `streamable` is set to `yes`, otherwise, it applies to any non-streaming document.
[5] The XSLT 3.0 Working Group is considering a proposal to limit an accumulator to a source document to prevent accumulators taking up too much processing time overhead. The status of this proposal can be tracked through XSLT 3.0 Bug 24547, read from comment#6.

An accumulator consists of three parts:

1. The top-level accumulator declaration `xsl:accumulator`, containing the name, the initial value, and whether or not is applies to streaming documents.
2. One or more accumulator rules, defined as children `xsl:accumulator-rule` elements of the accumulator declaration, which each defines a new value for the accumulator, whose calculation can be based on the previous value, obtainable through the reserved variable `$value`. An accumulator rule applies to a matching pattern, which follows rules equal to a match-pattern of a template[1]. There are two types of rules:
   - `phase="start"`, also called an *accumulator-before rule* applies the accumulator rule when the processor visits the opening tag[2], or beginning of the matching node. This means that, if you use the function `fn:accumulator-before` upon processing such node, it will return the new value just calculated in that rule. This is called the *pre-descent value*.

     You can request a pre-descent value only *before* a consuming instruction, or, if the body of a construct does not contain a consuming instruction, anywhere in that body.
   - `phase="end"`, also called an *accumulator-after rule*, applies the accumulator rule when the processor visits the closing tag, or end of the matching node. This means that, if you use the function `fn:accumulator-after` *after* you are done visiting that node, it will contain the new value calculated by the matching accumulator-after rule. This is called the *post-descent value*.

     You can request a post-descent value only *after* a consuming instruction, or, if the body of a construct does not contain a consuming instruction, anywhere in that body.
3. Zero or more usages of the accumulator, by the functions `fn:accumulator-before` and `fn:accumulator-after`, which take one parameter of type `xs:string`, whose value must match an existing accumulator name. When you can use which is described under the previous item.

Now that the essence of accumulators is clear, we can take a fresh look at the code. The streamable accumulator declared at the root and going by the name `count-words` and an initial value of `0` has one accumulator rule:

- *Accumulator-before rule:* it defines the `phase` as `start`, while this is the default, it is clearer to specify it explicitly.
- The match pattern is `text()[parent::para]`. This means that when this pattern matches any node, regardless of existing `xsl:apply-templates` instructions[3] or `xsl:for-each` loops, it will update the accumulator value attached to that node. Any accumulator matches twice, once at the start and once at the end of visiting a node (after processing its children). Left out phases default to leaving the current value unchanged.

  This pattern makes sure that we only match text nodes that are a child of `para` elements, which matches the original program listing. We cannot match `para` elements here, because then attempting to count the `text()` children would be considered non-motionless and is illegal in a streaming accumulator.
- The `new-value` attribute does what the `preceding::text()` expression did for us in the original code. At each visit of a text node, it adds the total count of the words to the previous count of the words.

  Allowing an expression on a text node may seem like a consuming expression and therefore not streamable. However, as we have seen in the first example in Section 4.9, “Rule 9: Use motionless filters”, expressions that consume childless nodes are considered motionless.

In the matching template for the `para` element, the code was changed to use this accumulator instead of the `following::` expression. This works, because on each time this template is processed, the current node will be a `para` element, and its children are not yet processed, which means that the running total on the first visit will be `0` (zero). Then, upon the call to `text()` in the `xsl:copy-of` instruction, the text node is processed and the running total is updated, which is reflected upon the next visit of the template. And so on, until the last `para` is visited.

[1] With two exceptions: you cannot match an attribute (you can reach out to attributes from an element with a motionless expression anyway), and you cannot match a namespace node (functions such as `in-scope-namespaces` are motionless, use them instead).

[2] This is correct: it is the only declaration that can depend on start- and end-tags, though these terms are not officially used in the specification, because it applies more generally to any node.

[3] Accumulators are global. They ignore the filter present in the `on-no-match` attribute, even if it is defined as `deep-skip`. If you want to skip nodes, you need to create a motionless pattern in the accumulator rules that does so. The only declaration that can have effect on accumulators is whitespace stripping, which happens prior to applying accumulators.

You might wonder how we can change this behavior and have the running total per paragraph include the count of the current paragraph, in other words, the first count should not be zero, but the total of that paragraph.

To achieve that, we need to make a few changes, but it is not as hard as it might seem at first:

- You might be tempted to change the accumulator rule phase to `phase="end"`, but this is not required in this scenario, because it operates on a childness node. That means that when the text node is processed, both the `fn:accumulator-before` and `fn:accumulator-after` functions can used within the matching template.
- Change the matching template to match the text node instead. Upon visiting the text node, the calculated value is available. This is different from our previous situation, where we requested the calculated value *before* the text node was visited. There is no *before* and *after* for a text node.
- Add the `parent::para` to text-matching template, otherwise we also match unwanted text-nodes.

The result of these changes looks as follows (only the changed parts are shown):

```
<xsl:template match="para">
  <xsl:copy>
    <xsl:apply-templates select="text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="text()[parent::para]">
  <xsl:attribute name="words-sofar"
    select="accumulator-before('count-words')"/>
  <xsl:copy-of select="."/>
</xsl:template>
```

5.7. Stylesheets requiring look-around

5.7.1. Intent

You have a stylesheet with expressions that rely on preprocessing the input tree and you want to make it streamable.

5.7.2. Level

Advanced, uses forking and multi-pass streaming. Streamability level: multi-pass[1].

5.7.3. Motivation

Consider you have been given the following input:

```
<expenses>
  <data date="2014-05-15">
    <amount article="pencils">28.97</amount>
    <amount article="paper">42.90</amount>
    <amount article="clips">18.41</amount>
    <amount article="ink">143.93</amount>
    <amount article="staples">6.23</amount>
  </data>
  <data date="2014-05-16">
    <amount article="pencils">44.62</amount>
    <amount article="paper">154.34</amount>
    <amount article="clips">6.19</amount>
    <amount article="ink">219.07</amount>
    <amount article="staples">0.00</amount>
  </data>
  <data date="2014-05-17">
    <amount article="clips">38.02</amount>
    <amount article="ink">108.71</amount>
    <amount article="staples">11.84</amount>
  </data>
</expenses>
```

and your requirement is to calculate daily totals and what the daily percentage is from the grand total, like this:

```
<expenses>
  <data total="240.44" perc="29" date="2014-05-15">
    <amount article="pencils">28.97</amount>
    <amount article="paper">42.90</amount>
    <amount article="clips">18.41</amount>
    <amount article="ink">143.93</amount>
    <amount article="staples">6.23</amount>
  </data>
  <data total="424.22" perc="52" date="2014-05-16">
    <amount article="pencils">44.62</amount>
    <amount article="paper">154.34</amount>
    <amount article="clips">6.19</amount>
    <amount article="ink">219.07</amount>
    <amount article="staples">0.00</amount>
  </data>
  <data total="158.57" perc="19" date="2014-05-17">
    <amount article="clips">38.02</amount>
    <amount article="ink">108.71</amount>
    <amount article="staples">11.84</amount>
  </data>
</expenses>
```

[1] Not all scenarios can be forked, it depends on whether it is possible to split processing where one part does not depend on intermittent output of another part. In other words, the forking pattern only works if you can process each part individually.

With XSLT 2.0, one could achieve it using a stylesheet like the following:

```
<xsl:template match="node() | @*">
  <xsl:copy>
    <xsl:apply-templates select="node() | @*"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="/expenses">
  <xsl:copy>
    <xsl:apply-templates select="node() | @*"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="data">
  <xsl:copy>
    <xsl:attribute name="total"
      select="sum(amount)"/>
    <xsl:attribute name="perc"
      select="round(sum(amount) div
              sum(//amount) * 100)"/>
    <xsl:apply-templates select="node() | @*"/>
  </xsl:copy>
</xsl:template>
```

On each visit of the the `data` element, we calculate the total of that day and the average in respect to the grand total. The total of that day is dependent of the children and the average is dependent on all `amount` elements, and as such, at the moment of visiting the `data` element, it requires (re)visiting all preceding and following `amount` elements.

Since looking around is not allowed in streaming, it is not immediately obvious how to make this example streamable. The first thing that comes to mind is using forking as in the *out of document order* pattern above. But that does not help us much, because we cannot store the result of one fork operation and use it in another for operation[1].

The streaming pattern discussed here will use an approach called multi-pass streaming, where the input stream is processed multiple times. It is not possible to that on any input stream, but if you know the document URI, you can open the streaming document multiple times using the `xsl:stream` instruction.

5.7.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Any expression requiring precalculation or preprocessing the input stream.
2. Expressions that cannot be split with forking.
3. Expressions that cannot use windowed streaming, for instance, that rely on the whole document.

Does not apply to

This pattern does not apply to:

1. Streaming documents that are not restartable[2].
2. Streaming documents that do not have an accessible URI.

5.7.5. Consequences

Applying this pattern has the following consequences:

1. Going twice or more over a large input document using streaming will increase the processing time by a factor of two or more.
2. Reliance on the document URI may have security implications.

5.7.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streaming mode.
2. Create a global variable to hold the initial input document URI.
3. Create a global variable for preprocessing the stream.
4. Apply previous patterns to make the stylesheet streamable.
5. Replace the free-ranging expressions with a variable reference.

[1] This is intentional in the design of `xsl:fork`, forking does not go over the input multiple times, instead, it visits each streamed input node one time as with normal streaming, but processes it multiple times, for each fork instruction.

[2] Sometimes an input document can only be processed *on the fly*, like a Twitter or news feed, a volatile memory stream or a listener to a network socket stream. Such streams cannot be restarted.

Example

Applying these five refactoring rules to our original example, we end up with the following code:

```
<!-- (1) introduce a streamable mode -->
<xsl:mode streamable="yes"/>

<!-- (2) capture the initial doc uri -->
<xsl:variable name="docuri"
    select="base-uri(.)"/>

<!-- (3) preprocess the total -->
<xsl:variable name="total">
  <xsl:stream href="{$docuri}">
    <xsl:value-of
      select="sum(//text()[parent::amount])"/>
  </xsl:stream>
</xsl:variable>

<xsl:template match="node() | @*">
  <xsl:copy>
    <!-- (4) apply streamability refactoring -->
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates select="node()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="/expenses">
  <xsl:copy>
    <!-- (4) apply streamability refactoring -->
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates select="node()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="data">
  <xsl:copy>
    <xsl:attribute name="total" select="$total"/>

    <!-- (4) apply streamability
             refactoring: forking -->
    <xsl:fork>
      <xsl:sequence>
        <!-- (5) use the precalculated $total -->
        <xsl:attribute name="perc"
          select="round(sum(amount) div
                  $total * 100)"/>
      </xsl:sequence>
      <xsl:sequence>
        <!-- (4) apply streamability
                 refactoring -->
        <xsl:apply-templates select="@*"/>
        <xsl:apply-templates select="node()"/>
      </xsl:sequence>
    </xsl:fork>
  </xsl:copy>
</xsl:template>
```

The example necessarily relies on previous streaming patterns, namely forking and splitting the identity select expression `node() | @*` into two; see previous sections for how these refactorings work.

The new streaming pattern introduced here is preprocessing the stream, or in other words, multi-pass streaming. There are no constructs in XSLT 3.0 that specifically facilitate multi-pass streaming, but using this pattern, it is relatively straightforward to do.

The variable `$docuri` relies on the global dynamic context. It is not allowed with streaming to consume a streamed node in a global variable, but it is allowed to use a motionless expression, in this case `base-uri(.)`, which stores the URI of the input document[1].

The second variable, `$total`, uses the `xsl:stream` instruction and the URI of the initial input document from `$docuri` to process the whole stream. Depending on the size of your stream, this can be a lengthy operation, but because variables are stable and deterministic in XSLT, this will be done only once.

The expression inside `fn:sum` may require a little explanation. The easier thing to do would be to write `//amount` as in the original non-streaming example. However, because of the chance of overlapping nodes, such an expression is not streamable within this function. This limitation may be lifted in streamability analysis, in fact there is a strong sentiment in the Working Group to allow such constructs and to allow limited buffering by processors, but at the time of this writing, there was no decision yet. We have seen before that childless nodes do not suffer the same problem (there is no chance for overlap), hence we can rewrite the expression to focus on the the text nodes as `text()[parent::amount]`, which is streamable.

After defining a global variable for preprocessing the input stream, it becomes relatively trivial to rewrite the rest of the stylesheet to use this variable and to introduce forking for the part that still have multiple downward selects.

Note: processing a streaming document is by definition not stable, because the document will not be held in memory. That means, if during streaming the document's content changes, this will effect the outcome of the transformation. Going over the document twice, as in this streaming pattern, has the same potential side-effect: the document may change between processing through the `xsl:stream` instruction and the initial input tree processing in the streaming mode. It is up to the document provider to make sure that the document remains stable for the duration of the processing.

[1] The rules on accessing the global context node in a streaming context is still under debate in the working group. It is possible that support for this will be dropped, in which case you should use a parameter instead and pass the document URI on through the infrastructure of your processor.

5.8. Dependencies on xsl:call-template

5.8.1. Intent

You have a stylesheet with dependencies on `xsl:call-template` and want to make it streamable.

5.8.2. Level

Intermediate, sometimes requires forking. Streamability level: full.

5.8.3. Motivation

Consider the following example:

```
<xsl:template match="person">
  <xsl:call-template name="address">
    <xsl:with-param name="street"
                    select="address/street"/>
    <xsl:with-param name="number"
                    select="address/number"/>
  </xsl:call-template>
</xsl:template>
<xsl:template name="address">
  <xsl:param name="street"/>
  <xsl:param name="number"/>
  <xsl:value-of select="$number, $street"/>
</xsl:template>
```

The instruction `xsl:call-template` is severely limited when used with streaming. You cannot pass references to nodes and you cannot use the context item. If there is an implicit context item as in the example above, streamability analysis will fail.

5.8.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Any use of `xsl:call-template` involving streamed nodes.
2. Any other `xsl:call-template`, unless the context item is explicitly prohibited.

Does not apply to

This pattern does not apply to:

1. Other instructions than `xsl:call-template`.

5.8.5. Consequences

Applying this pattern has the following consequences:

1. Your programmers will have to learn not to use `xsl:call-template` in streaming scenarios.

5.8.6. Implementation

There are no existing situations that I know of that actually *require* the use of `xsl:call-template`. The streaming pattern proposed here is to get rid of it. However, there is one scenario where this is not necessary, which is when the called template is not dependent on the context item and the parameters are motionless expressions. In such cases, simply add a `<xsl:context-item use="prohibited"/>`[1] to the called template.

If there is dependency on streaming nodes, refactor your original code by applying the following steps in order:

1. Introduce a streaming mode.
2. Replace the `xsl:call-template` instructions with either an `apply-templates` (possibly in another named, but streamable, mode) or a stylesheet function call.
3. Remove dependencies on streaming nodes in parameters.
4. Apply other patterns where necessary to make the result streamable.

[1] The instruction `xsl:context-item` is new in XSLT 3.0 and determines the type of the expected context item and whether or not there should be a context item at all. Using `use="prohibited"` prohibits a context item, which, in the case of `xsl:call-template` means that the context item will not be passed on to the template, it will be absent.

Example

Applying these four refactoring rules to our original example, we end up with the following code:

```
<!-- (1) change the mode to streamable -->
<xsl:mode streamable="yes"/>

<xsl:template match="person">
  <!-- (2) replace call-template
           with apply-template -->
  <xsl:apply-templates name="address"/>
</xsl:template>

<!-- (2) replace named template -->
<!-- (3) remove parameters -->
<xsl:template match="address">
  <xsl:fork>
    <!-- (4) apply other
             patterns, here: forking -->
    <xsl:sequence select="string(number) || ' '"/>
    <xsl:sequence select="string(street)"/>
  </xsl:fork>
</xsl:template>
```

In most cases, removing dependencies on `xsl:call-template` will be rather trivial and only requires using known XSLT 2.0 constructs. In this example, we had two downward expressions (potentially) not in document order, which forced us to use forking as explained in an earlier streaming pattern. Other than basic refactoring to get rid of the named template, there are no new concepts revealed here.

5.9. Using streamable stylesheet functions

5.9.1. Intent

You have a stylesheet function with arguments taking nodes and you want to make it streamable.

5.9.2. Level

Advanced, requires understanding writing streamable functions. Streamability level: full.

5.9.3. Motivation

Consider the following example:

```
<xsl:template match="price">
  <xsl:value-of select="f:vat(.)"/>
</xsl:template>

<xsl:function name="f:vat">
  <xsl:param name="price"/>
  <xsl:value-of select="($price div 121) * 21)"/>
</xsl:function>
```

It shows a simple function that takes a `price` element and calculates the VAT from a VAT-inclusive price, here 21%.

5.9.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Stylesheet functions that have at most one parameter that can be a node.
2. Stylesheet functions returning streamed nodes.

Does not apply to

This pattern does not apply to:

1. Stylesheet functions that take more than one parameter that can be node[1].
2. Recursive stylesheet functions[2].
3. Non-final stylesheet functions[3].

5.9.5. Consequences

Applying this pattern has the following consequences:

1. All your stylesheet functions signatures will be typed.
2. Streamable stylesheet function bodies will need to be and remain at most consuming (one downward select, see Section 4.1, “Rule 1: each template rule can have a maximum of one downward expression”), which has consequences on maintainability.

[1] The brand new feature that allows functions to take streamable nodes allows them to have at most one parameter that is a streamable node. This is defined in the signature of the function.

[2] It is possible to write functions that are recursive and guaranteed streamable, but it is a very advanced concept, very new and likely to change, and out of the scope of this paper.

[3] In XSLT 3.0 it is possible to use packages (declared with `xsl:package`) and to override modes, named templates and functions, as long as they are marked overridable. It is currently not allowed to use streamable functions that are not final, you must either override a non-final function and make it final, or you must change the function signature to be final.

5.9.6. Implementation

Refactor your original code by applying the following steps in order:

1. Mark the stylesheet function as streamable (if you have not already done so, introduce a streamable mode as well).
2. Type the return type and the function parameters.
3. Change the function body to be streamable using other streamable patterns.

Example

Applying these three refactoring rules to our original example, we end up with the following code:

```
<!-- (1) introduce a streamable mode -->
<xsl:mode streamable="yes"/>

<xsl:template match="price">
  <xsl:value-of select="f:vat(.)"/>
</xsl:template>

<!-- (2) type the function's return type -->
<xsl:function name="f:vat" as="xs:string">
  <!-- (2) type the function's parameters -->
  <xsl:param name="price" as="element()"/>

  <!-- (3) make body streamable,
           here: nothing to do -->
  <xsl:value-of select="($price div 121) * 21)"/>
</xsl:function>
```

An important step here is to type the parameters and the return type. It is allowed and possible to create streamable functions without it, but it is much harder to write streamable functions that way. By telling the processor that the return type is an atomic type, it can analyse that the function will consume the input node and it can mark calling the stylesheet function with *usage absorption*.

Inside the function's body, we have an atomizing construct: the variable reference `$price`, which is bound to a potentially streaming node, will be atomized before division takes place. Atomization always means that the node is going to be consumed. Function declarations follow the same rules as templates, Section 4.1, "Rule 1: each template rule can have a maximum of one downward expression". However, within functions, the bound parameter is the streaming node for the sake of the analysis. In other words, the call on the `$price` parameter is now the replacement for the downward select (you could mentally replace it with a node reference, such as the `self::price` select expression to see how it works).

Streamable stylesheet functions are a very recent addition to the machinery available to authors of streamable stylesheets and packages. It is a very powerful one, but it is also one of the most complex to write correctly. It is the only user-defined way of writing a construct that returns nodes which is allowed (recall that named templates cannot operate on streaming nodes at all and that templates are always grounded, meaning that they can never return nodes).

For instance, it is possible to write a function that returns the third child based on a certain pattern. The returned node will still be a streaming node and not, as with other constructs, a copy of a node:

```
<xsl:function name="f:child"
              streamable="yes"
              as="element()">

  <xsl:param name="node" as="element()"/>
  <xsl:sequence
    select="$node/person[@gender = 'M'][3]"/>
</xsl:function>
```

If you call that function with a streamable expression, you can apply path expressions on the returned value and you will operate on the original node, not a copy of the node. Besides from not requiring the overhead of copied nodes, it has the advantage that the identity of the nodes is preserved. An example of a valid expression is: `member/f:child(.)/age`.

5.10. Sorting

5.10.1. Intent

You have a stylesheet construct that requires sorting and you want to make it streamable.

5.10.2. Level

Advanced, requires two-phase streaming, streamable grouping and streamable merging. Streamability level: none[1] and then: implicitly windowed.

5.10.3. Motivation

Consider the following example:

```
<xsl:template match="log-entry">
  <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="/log">
  <xsl:copy>
    <xsl:apply-templates select="log-entry">
      <xsl:sort select="@date"/>
    </xsl:apply-templates>
  </xsl:copy>
</xsl:template>
```

It takes an unsorted log document and sorts it by date. The input could, for instance, look something like the following:

```
<log>
  <log-entry
    date="2014-05-10">Some log text</log-entry>
  <log-entry
    date="2014-05-07">Some log text</log-entry>
  <log-entry
    date="2014-05-06">Some log text</log-entry>
  <log-entry
    date="2014-05-06">Some log text</log-entry>
  <log-entry
    date="2014-05-04">Some log text</log-entry>
  <log-entry
    date="2014-05-07">Some log text</log-entry>
  <log-entry
    date="2014-05-07">Some log text</log-entry>
  <log-entry
    date="2014-05-07">Some log text</log-entry>
  <log-entry
    date="2014-05-02">Some log text</log-entry>
  <log-entry
    date="2014-05-09">Some log text</log-entry>
</log>
```

5.10.4. Applicability

Applies to

This pattern applies to the following scenarios:

1. Any construct using sorting.

Does not apply to

This pattern does not apply to:

1. Sorting where the sortable items easily fit in memory, in such scenarios, use a copy of the nodes and apply the sorting to this copy.

5.10.5. Consequences

Applying this pattern has the following consequences:

1. Changes the infrastructure for multiple processing, you should consider adding [XProc] to the toolchain.
2. The sorting itself will not be obvious in either of the stylesheets.
3. Requires temporary disk space, equal to the size of the input document, to store the intermediate results.

5.10.6. Implementation

Refactor your original code by applying the following steps in order:

1. Introduce a streamable mode in the original stylesheet.
2. Use streamable grouping to create chunks of the input data, use `group-adjacent`.
3. Use `xsl:result-document` to write each groups to disk.
4. Create a copy of each chunk and apply the sorting on the copy, this part need not be streamable[2].
5. Apply any necessary refactoring to make the rest of the stylesheet streamable.
6. For phase two, create another streamable stylesheet
7. Get a collection of the URIs from the documents created in step 3.
8. Add streamable `xsl:merge` instruction to process this collection.

[1] Sorting itself is not streamable at all. The solution provided for the streamable design pattern is to split the source document into manageable chunks, sort those chunks and then process those again using streamed merging.

[2] As with other streamable design patterns, there are no limitations on expressions applied to copies of nodes.

Example

Applying these eight refactoring rules to our original example, we end up with two stylesheets, the first, which creates the smaller sorted chunks of the input document, looks as follows:

```
<!-- (1) introduce a streamable mode -->
<xsl:mode streamable="yes"/>

<!-- (5) other refactoring, here: nothing to do -->
<xsl:template match="log-entry">
  <xsl:copy-of select="."/>
</xsl:template>

<xsl:template match="/log">
  <!-- (2) use streamable grouping -->
  <xsl:for-each-group
    select="log-entry"
    group-adjacent="position() idiv 3">

    <!-- (3) write chunks to disk -->
    <xsl:result-document
      href="sorted_{current-grouping-key()}.xml">
      <log>
        <!-- (4) create copy of each chunk -->
        <xsl:variable name="copy" as="element()*"
           select="copy-of(current-group())"/>

        <!-- (4) and apply sorting on the copy -->
        <xsl:apply-templates select="$copy">
          <xsl:sort select="@date"/>
        </xsl:apply-templates>
      </log>
    </xsl:result-document>
  </xsl:for-each-group>
</xsl:template>
```

Most of the rewriting here follows streaming design patterns seen in previous sections. Some parts warrant extra attention:

Using `group-adjacent` is allowed in streamable scenarios as long as the expression is motionless. Using `fn:position` is motionless, and so is any calculation on it. With `idiv` we get nice round numbers for our splitting. The size for our chunks here is 500, but any number can be used as long as the chunks by themselves fit in memory.

A copy of each group is created using `fn:copy-of`, and stored in a variable. Instead of storing it in a variable, you can inline this in the apply-templates instruction[1].

Finally, applying templates on this copy and sorting them using `xsl:sort` is allowed, because the nodes are not streamed anymore, they are a copy. The streaming design pattern so far, without the `xsl:result-document` and the grouping, can serve as a solution if the nodes to be sorted fit in memory.

The stylesheet for the second phase looks as follows:

```
<!-- (6) 2nd pass streamable stylesheet -->
<xsl:mode streamable="yes"/>

<xsl:template match="/log">
  <xsl:copy>
    <!-- (7) collection of uris,
             see your proc's documentation -->
    <xsl:variable name="uricoll"
      select="uri-collection('sorted-chunks')"/>

    <!-- (8) merge sorted output from 1st phase -->
    <xsl:merge>
      <xsl:merge-source
        for-each-stream="$uricoll"
        select="log/log-entry">
          <xsl:merge-key select="@date"/>
      </xsl:merge-source>
      <xsl:merge-action>
        <xsl:copy-of
          select="current-merge-group()"/>
      </xsl:merge-action>
    </xsl:merge>
  </xsl:copy>
</xsl:template>
```

An `xsl:merge` instruction takes one or more input documents and merges them based on a merge-key, similar to the way sorting works. It processes the input documents one item at the time, based on the select-statement in the `xsl:merge-source` instruction. In the case of streaming, it does so in order, which is why the input documents must be pre-sorted. It will output the elements in the order defined by the merge-key, inserting the elements from the other source documents where necessary, to preserve order.

The input for `xsl:merge-source` is given by the `for-each-stream` attribute, which takes a sequence of strings that must be valid URI's. In the above example, I use `fn:uri-collection`, which works like `fn:collection`, but instead of returning a sequence of document nodes, it will return a sequence of URI's. The way a collection URI is interpreted is implementation dependent, which is why in the above example I wrote "see the processor's documentation". Most implementations will support a kind of wildcard matching and the usage of some form of XML catalogs.

[1] Some versions of Saxon did not allow multiple nodes as argument to `fn:copy-of`, however, the specification does allow this.

The `xsl:merge-action` defines the action to occur upon each merge operation. There will be one action for each unique merge-key. If multiple elements have the same key, they are added to the current merge group, which can be requested, similar as with grouping, using `fn:current-merge-group`. The current merge group is implicitly grounded and maintains access to its ancestors similar to a call to `fn:snapshot`. This has the advantage that you can use free-ranging expressions on the merge group.
In our scenario, no additional processing has to take place. We simply want all the `log-entry` elements merged back into one big document. The result is a sorted document.
This method of sorting, using splitting, sorting on small chunks and then merging back is the simplest way to do sorting in a streaming way. It is also the only way to do streamable sorting that requires only two phases. A single phase streaming sort is not possible.

6. Streamable packages

With XSLT 3.0 a new feature called *packaging* has been introduced. Packaging allows authors to create libraries of stylesheet functions, stylesheet templates and other stylesheet declarations that can be precompiled, distributed and used as a library of functions with other stylesheets. It extends the ways `xsl:import` works, it allows a certain level of overridability similar to object-oriented languages and it supports information hiding through private and public named declarations.

Package authors that wish to reach a largest as possible audience for their library packages, would want to prepare their packages to behave properly in streaming scenarios. Since non-streaming scenarios can safely ignore the streamability properties, it is suggested that package authors strive to write their packages, the public stylesheet functions and public modes with streamability in mind.

It is not possible to create a public mode to be both streamable and non-streamable. But it is possible to create two modes, one streamable and one not, that is processed by the same streamable templates that each have their mode set to `#all` or at least to both modes.

For stylesheet functions, they can safely have the `streamable` attribute set to `yes` if they take nodes as parameters. A non-streamable usage of a streamable function behaves without side effects exactly the same as in a streamable context. If package authors choose to write defensively and write each function according to the guaranteed streamability rules, their packages can be used cross-processor and with both streamable and non-streamable input.

7. Conclusion

Many common classical XSLT programming scenarios and patterns appear to be convertible into guaranteed streamable code with relative ease. Following ten easy-to-remember rules, with on top of the list the rule on having a maximum of one downward expression per context or template brings us closer to a fully streamable stylesheet. Taking original programming patterns and seeing how they change into streamable programming patterns should give the intermediate to advanced XSLT programmer, or the beginning Streaming XSLT programmer a good starting point.

Some more complex scenarios including forking, streamable grouping, merging and even sorting were covered, showing that the current state of the specification, including the fixes available through BugZilla, work with a wide range of potential streaming use-cases.

Streaming is not hard, you just need to set your mind to it, and once set, adjusting your code to process large, even huge documents, becomes almost a breeze.

Bibliography

[BRA14] *Streaming for the masses*. Abel Braaksma. XML Prague 2014. ISBN 978-80-260-5712-3. http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf

[BUGZ] *Bugzilla - Public W3C Bug / Issue tracking system*. 2014. Miscelleneous authors. https://www.w3.org/Bugs/Public/

[FO3] *XPath and XQuery Functions and Operators 3.0, latest version*. 2014. Michael Kay. http://www.w3.org/TR/xpath-functions-30/

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

[MKAY08] *XSLT 2.0 and XPath 2.0 Programmer's Reference*. 4th edition. 2008. Michael Kay. ISBN: 978-0-470-19274-0

[XDM] *XQuery and XPath Data Model 3.0, latest version*. 2014. Norman Walsh, Anders Berglund, and John Snelson. http://www.w3.org/TR/xpath-datamodel-30/

[XP3] *XML Path Language (XPath) 3.0, Latest Version*. 2014. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. http://www.w3.org/TR/xpath-30/

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

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

[XSLT3] *XSL Transformations (XSLT) Version 3.0, Latest Version*. 2013. Michael Kay. http://www.w3.org/TR/xslt-30/

[XSLWD] *XSL Transformations (XSLT) Version 3.0, W3C Last Call Working Draft 12 December 2013*. 2013. Michael Kay. http://www.w3.org/TR/2013/WD-xslt-30-20130201/

From monolithic XML for print/web to lean XML for data: realising linked data for dictionaries

Matt Kohl

Oxford University Press

<matt.kohl@oup.com>

Sandro Cirulli

Oxford University Press

<sandro.cirulli@oup.com>

Phil Gooch

Oxford University Press

<phil.gooch@oup.com>

Abstract

In order to reconcile the need for legacy data compatibility with changing business requirements, proprietary XML schemas inevitably become larger and looser over time. We discuss the transition at Oxford University Press from monolithic XML models designed to capture monolingual and bilingual print dictionaries derived from multiple sources, towards a single, leaner, semantic model. This new model reflects the lexical content units of a traditional dictionary, while maximising human readability and machine interpretability, thus facilitating transformation to Resource Description Framework (RDF) triples as linked data.

We describe a modular transformation process based on XProc, XSLT, XSpec and Schematron that maps complex structures and multilingual metadata in the legacy data to the structures and harmonised taxonomy of the new model, making explicit information that is often implicit in the original data. Using the new model in its prototype RDF form, we demonstrate how cross-lingual, cross-domain searches can be performed, and custom data-sets can be constructed, that would be impossible or very time-consuming to achieve with the original XML content stored at the individual dictionary level.

Keywords: XProc, RDF, dictionaries

1. Introduction

Oxford University Press (OUP) publishes a number of academic dictionaries both in print and online, most notably the Oxford English Dictionary [1], as well as current dictionaries for English and bilingual dictionaries for modern languages [2]. Recently, we have also expanded our dictionaries offering by licensing in lexical content from other publishers as part of the Oxford Global Language Solutions (OGLS) [3] initiative, enabling us to meet demands for lexical content in languages outside OUP's catalogue.

As there is no standard for digital publication of dictionaries, the OGLS data-sets come to us in a variety of formats (XML, MySQL, InDesign, MS Word and Excel, plain text, etc.) To offer this licensed-in content to our customers, we draw on in-house and freelance developers to convert to OxMonolingML and OxBilingML, our dictionary XML Document Type Definitions (DTDs). These DTDs evolved from SGML and were originally designed to capture a dictionary as it appears on a printed page. Recently, we have received feedback from customers about excessive hierarchy and complication in these models. This added to a growing concern within our team that the DTDs have become somewhat monolithic and permit too much variation, a consequence of frequently needing to loosen the structures to capture content from diverse print sources. Despite our attempts to combat permissiveness with best-practice advice and supplement with Schematron validation [4], the XML we receive from our developers often departs from our standards.

Meanwhile, a pilot project to create linked data from a sample of dictionary data-sets was approved. The proposal involved designing a schema to model language concepts, rather than print conventions, one which results in intelligent, semantic, machine-interpretable lexical content in XML.

2. Data Modelling

While the OxMonolingML & OxBilingML DTDs capture the formatting intricacies of print dictionaries and enable these to be reproduced online, they are not ideal data models for machine processing. The rationale for developing a new dictionaries data model was two-fold. First, the need to future-proof, as far as possible, our multilingual lexical content, so that it will be reusable and interoperable with other tools and data-sets in the growing fields of language technology and linked data. Second, to address licensee feedback about the difficulty of understanding and processing a deeply hierarchical, and overly complicated content model that tended to lack semantic consistency across data-sets. The following requirements were prioritised:

- *Self-documenting*: Data structures, element names and attribute values should be human understandable.
- *Semantically rich*: Data structures should be readily machine-processable and machine-interpretable.
- *Well structured*: There should be only one, clear way to model any given lexical item.

The third requirement was particularly important, as the OxMonolingML and OxBilingML DTDs allow optional and repeatable structures in many places while lacking clear semantics, resulting in many ways of modelling the same lexical structure. The DTDs also permit mixed content (elements that may contain other elements, text, or both) in many places. Without additional Schematron validation, this caused numerous quality assurance problems, as many elements could be validly empty (according to the models), but invalid semantically. For example, does the appearance of empty element content in a mixed content model imply a data conversion error, editorial omission, or genuine lack of data? The new model needed be expressed in a more rigorous formalism to reduce the need for numerous Schematron content validation rules.

From these requirements, the following modelling principles were determined:

- The model should be agnostic to source or language.
- The model should be able to handle both monolingual and bilingual data.
- The model should be modular, to facilitate maintainance and customisation.
- Attribute values, as far as possible, should come from controlled vocabularies.
- Mixed content should be reduced to a minimum.
- The content should be readily transformed to other structured data, such as RDF.

As a result of being strict on mixed content, we decided that empty elements should have a specific semantic purpose - empty element content in XML instances must not be used to imply lack of content. For example, initially we decided that a word sense in a monolingual dictionary should always contain a definition. However, cases later arose where this could not always be enforced. Rather than make `<definition>` optional or allow it to be empty, we introduced a `<noDefinition>` element to be used as a placeholder. Any empty or missing `<definition>` elements in an instance would therefore be a data conversion error.

Additional modelling principles arose from the need to simplify the complex sense hierarchy, (which was in any case inconsistent across dictionaries) and our goal of making the data more modular and less monolithic:

- Subsidiary relationships should be preserved via attribute values pointing to the identifier of the parent sense, rather than traditional nesting, effectively flattening the sense XML without sacrificing an entry's complexity.
- The scope of an entry should always be a single Lemma. Derivatives, phrases, and other items traditionally nested in sub-entries are all upgraded to entry level. Again, formerly hierarchical relationships are retained through semantic linking.

DTDs remain popular at the Press as they are familiar and understandable to both editors and content developers. However, to facilitate enhanced content validation and maximise interoperability and tool support, we selected XML Schema as the primary formalism for the new model. Because a dictionary entry can also be represented as a collection of metadata about a word or phrase, we undertook parallel modelling in Resource Description Framework (RDF) [5], Web Ontology Language (OWL) [6] and XML Schema.

Both the document-centric and metadata models consider the core concepts of Lemma (dictionary headword), Etymology (origin), Sense (semantics), Definition (meaning), Example (usage), and their relationships. The XML model reflects the lexical content units of a traditional dictionary entry, and so explicitly models concepts of synonymy and meronymy (phrasal constructions composed of one or more lemmas) as specific element types that indicate their relationship to the lemma. The RDF model goes further and considers these as lemmas that happen to be in a synonym or meronym relationship. For example, in the XML model we have

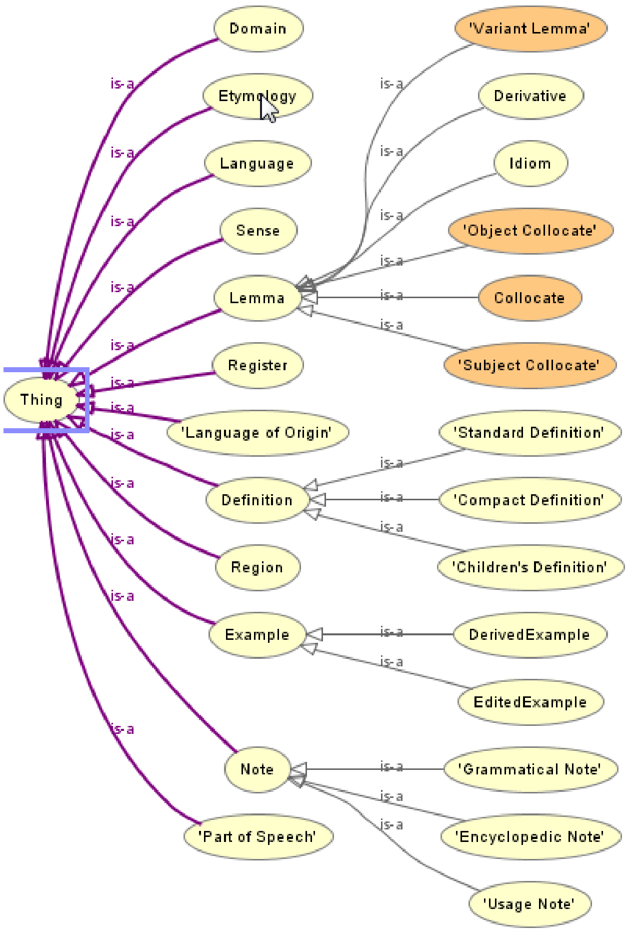
```
<sense id="d5e644">
  <definitions>
    <definition id="d5e649" xml:lang="en">
      <text>
        An English definition of the term
      </text>
    </definition>
  </definitions>
  <synonyms>
    <synonym targetid="a5e644">
      another term
    </synonym>
  </synonyms>
</sense>
```

whereas in the RDF model the same relationships are reified and would be represented as

```
<Sense rdf:about="sense:d5e644">
  <hasSynonym rdf:resource="lemma:a5e644"/>
  <hasDefinition rdf:resource="definition:d5e649"/>
</Sense>
<Definition rdf:about="definition:d5e649">
  <rdfs:label xml:lang="en">
    An English definition of the term
  </rdfs:label>
</Definition>
<Lemma rdf:about="lemma:a5e644">
  <rdfs:label xml:lang="en">
    another term
  </rdfs:label>
</Lemma>
```

The conceptual classes are shown in Figure 1, “Reified conceptual classes”.

Figure 1. Reified conceptual classes



Instances of the Register, Domain, PartOfSpeech classes were modelled as hierarchical, controlled vocabularies in RDF, with translations in major languages. For example:

```
<Domain rdf:about="domain:physics">
  <rdfs:subClassOf
    rdf:resource="domain:physical-science"/>
  <rdfs:label xml:lang="en">Physics</rdfs:label>
  <rdfs:label xml:lang="de">Physik</rdfs:label>
  <rdfs:label xml:lang="es">Física</rdfs:label>
</Domain>
```

This provided a mechanism to map metadata values in the legacy data to language-independent identifiers in the new model (see Section 3, “Data Conversion”).

3. Data Conversion

Our dictionaries catalogue includes more than 40 monolingual and bilingual titles, and to move that much data into the new model, we would need a scalable transformation strategy. Furthermore, conversion would require some standardisation steps that could be leveraged to enhance the OxMonolingML & OxBilingML data for our current customers, so building something modular was also desirable.

Given these requirements, we decided to build a modular XProc [7] pipeline comprising XSLT, Schematron and XML Schema validation steps, with specific modules for each language in order to hold harmonised metadata across dictionaries and languages. A simplified diagram of the XProc pipeline is shown in Figure 2, "XProc pipeline".

The input port of the XProc pipeline is a monolingual dictionary in the OxMonolingML DTD format. Three parameters may be passed to the pipeline in order to apply language-specific or licensee-specific steps as needed.

The input file goes through a sequence of XSL transformations which yield an intermediate file that we like to call OxMonolingML++. This is an enhanced version of the source data, featuring harmonised structural variations in sub-entries, synonyms, antonyms, etc. as well as metadata mappings from the source language to their English equivalent, e.g.:

```
<pos value="adjective">adjetivo</pos>
```

These steps are grouped inside a single sub-pipeline ending with the Schematron validation. The sub-pipeline has dedicated output ports for storing the OxMonolingML++ file and the Schematron error report.

A similar process is applied in the second sub-pipeline that generates the final output referred in the diagram as Lexical Data. This sub-pipeline also allows for licensee-specific steps triggered by the corresponding licensee parameter.

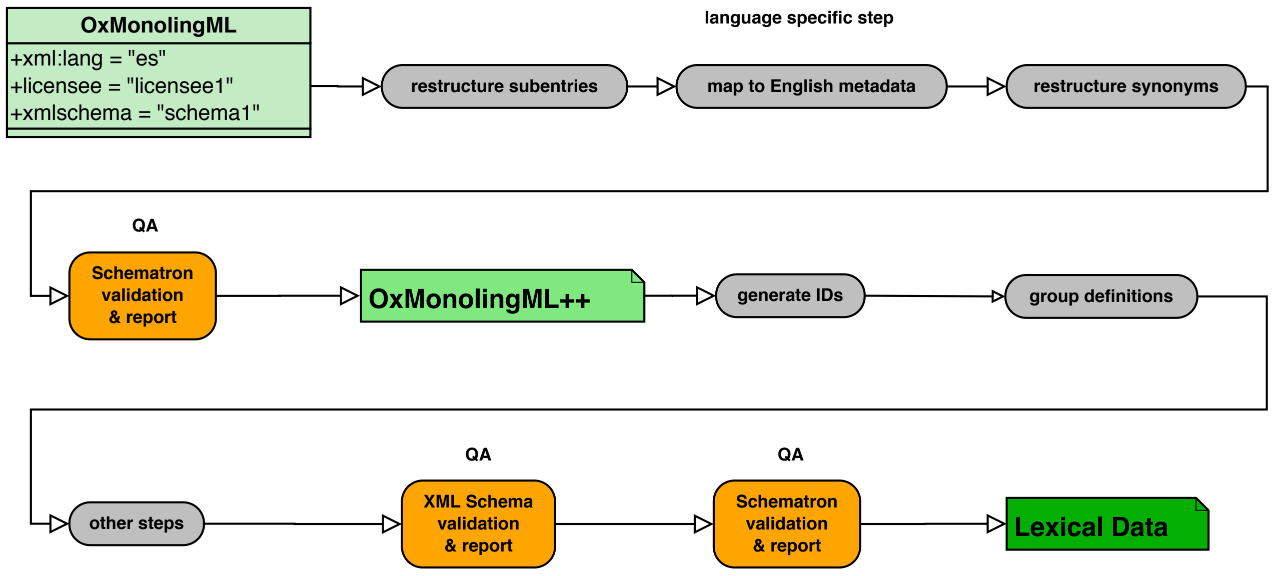
Both Schematron and XML Schema validations are performed on the final output with dedicated output ports for storing error reports. When there are licensee-specific steps, an XSL transformation generates a corresponding version of the XML schema for validation of the custom output.

Development of the pipeline components was carried out by a team of four developers and a project manager using Agile methodologies. Behaviour-driven development was employed by writing XSpec unit tests [8] for core transformation scenarios, as illustrated in Example 1, "XSpec unit test".

Example 1. XSpec unit test

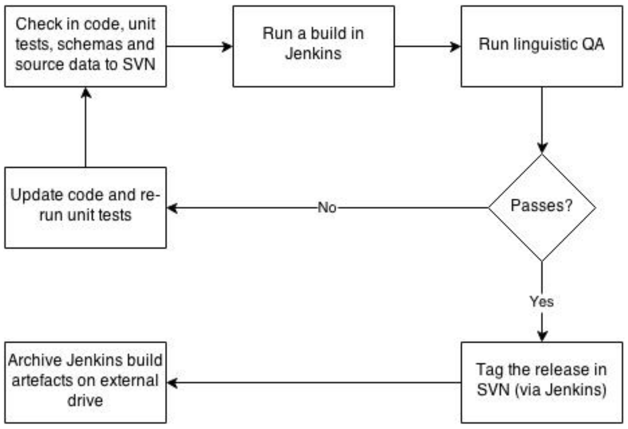
```
<x:scenario label="When processing adj inv/adv">
  <x:context>
    <pos>adj inv/adv</pos>
  </x:context>
  <x:expect label="It should produce an
                   adjective and an adverb">
    <pos value="adjective">adj inv</pos>
    <genpunc>/</genpunc>
    <pos value="adverb">adv</pos>
  </x:expect>
</x:scenario>
```

Figure 2. XProc pipeline



Jenkins [9] was employed as a continuous integration server for running automated XSpec unit tests and building the final deliverables both for licensees and internal use. The workflow is illustrated in Figure 3, "Build workflow".

Figure 3. Build workflow


Check in code, unit tests, schemas and source data to SVN
Run a build in Jenkins
Run linguistic QA
Passes?
No
Update code and re-run unit tests
Yes
Tag the release in SVN (via Jenkins)
Archive Jenkins build artefacts on external drive

XML source data in OxMonolingML, XSL and XSpec code, Schematron and XML Schema are managed under Subversion (SVN) [10]. The Jenkins build process checks out the latest version of the code and data from the repository, and then runs the main conversion pipeline via Ant [11], which executes the XSpec unit tests, generates and executes the XProc pipeline, and runs additional tasks for packaging the deliverables. The artefacts created by the build process are reviewed by a linguist for quality assurance. If any critical errors arise from unit tests or linguistic QA, the code is fixed through another iteration. Otherwise, Jenkins commits and tags the final deliverable in SVN with a release number, and archives it on an external drive in order to be shipped to the licensee.

The build process can also be run in 'test' mode, triggered when any code or data changes are made to the repository. Access to the latest unit test results are always available from the Jenkins web interface.

For better scalability (and to avoid the need to maintain a single, monolithic pipeline), the framework relies on pipeline configuration files, which outline the required steps and metadata mappings for specific data-sets. Using these configurations and some boilerplate pipeline XSL, the Ant script builds and executes a custom XProc on the fly.

4. Results and Discussion

The following lists provide an overview of the data at each stage of our transformation.

Example 2, "Spanish entry for abanicar in original source data" shows a Spanish dictionary entry in the format licensed from the original publisher, i.e. before it has been converted to our OxMonolingML DTD by one of our freelance or in-house developers. The structure is very straightforward, but all content, including element and attribute names, is in the source language, making interoperability and reuse more difficult.

Example 2. Spanish entry for abanicar in original source data

```
<ENTRADA ID="370" ORDER="370" OLDID="350">
  <ZONA-ENTRADA>
    <LEMA>abanicar</LEMA>
  </ZONA-ENTRADA>
  <CATGRAM>&verbo_transitivo_dueae;</CATGRAM>
  <ACEPCIO ACEP="1">
    <SIGNIFICAT>
      Dar aire con el abanico u otro objeto:
    </SIGNIFICAT>
    <EXEMPLE>
      Pedro se abanica descuidadamente
      con su sombrero de copa y muestra un gesto
      de aburrimiento.
    </EXEMPLE>
  </ACEPCIO>
  <ACEPCIO ACEP="2">
    <TEMA>taur</TEMA>
    <SIGNIFICAT>
      Incitar al toro agitando ante él el capote de
      un lado a otro, generalmente para que cambie
      de lugar en la suerte de varas.
    </SIGNIFICAT>
  </ACEPCIO>
  <OBSERVACIO NUM="1" CLASE="conjugacion"
              TIPO="irregular">
  Conjug. [1] como <C>sacar</C>.</OBSERVACIO>
</ENTRADA>
```

Example 3, “Spanish entry for abanicar in OxMonolingML” shows the same Spanish data following conversion to the OxMonolingML DTD. Metadata is still in Spanish at this point, and the structure is arguably less transparent than the original (due to the OxMonolingML DTD needing to capture a wide range of content for both print and online use). We used this data as input for our conversion framework.

Example 3. Spanish entry for abanicar in OxMonolingML

```
<e xrid="370" type="standard">
  <hg>
    <hw>abanicar</hw>
  </hg>
  <sg>
    <se1>
      <posg>
        <pos>verbo transitivo</pos>
      </posg>
      <se2 num="1">
        <msDict type="core">
          <df>Dar aire con el abanico u otro
objeto<genpunc tag="SIGNIFICAT">:</genpunc></df>
          <eg>
            <ex>Pedro se abanica descuidadamente con su sombrero
de copa y muestra un gesto de
aburrimiento<genpunc tag="EXEMPLE">.</genpunc></ex>
          </eg>
        </msDict>
      </se2>
      <se2 num="2">
        <lg>
          <sj>taur</sj>
        </lg>
        <msDict type="core">
          <df>Incitar al toro agitando ante él el capote de un lado
a otro, generalmente para que cambie de lugar en la suerte de
varas<genpunc tag="SIGNIFICAT">.</genpunc></df>
        </msDict>
      </se2>
    </se1>
  </sg>
  <note type="conjugacion|irregular" position="entry">Conjug. [1]
  como <i>sacar</i><genpunc tag="OBSERVACIO">.</genpunc></note>
</e>
```

Example 4, “Spanish entry for abanicar in new Lexical model” shows the output following XSLT transformation to the new Lexical model. All metadata are now from controlled vocabularies (e.g. `partOfSpeech="verb"` and `domain="bullfighting"`), and the structure is more transparent (NB conjugation information has been omitted, as we hold language data on constructions, conjugations and morphology in a separate repository).

Example 4. Spanish entry for abanicar in new Lexical model

```
<entry id="d5e1674">
  <heading xml:lang="es" partOfSpeech="verb">
    <headword>abanicar</headword>
  </heading>
  <senses>
    <sense id="d5e1683">
      <definitions>
        <definition xml:lang="es">
          <text>Dar aire con el abanico u otro objeto</text>
        </definition>
      </definitions>
      <examples>
        <example>
          <text>Pedro se abanica descuidadamente con su sombrero
          de copa y muestra un gesto de aburrimiento</text>
        </example>
      </examples>
    </sense>
    <sense domain="bullfighting" id="d5e1694">
      <definitions>
        <definition xml:lang="es">
          <text>Incitar al toro agitando ante él el capote de
          un lado a otro, generalmente para que cambie de lugar
          en la suerte de varas</text>
        </definition>
      </definitions>
    </sense>
  </senses>
</entry>
```

Example 5, "Spanish entry for abanicar in RDF/XML" shows the output following XSLT transformation of the Lexical model to RDF/XML. Controlled vocabulary labels have been replaced with compact URIs using CURIE syntax [12] pointing to instances in the domain ontology. This allows us to use inference when writing SPARQL queries to interrogate the RDF data. For example, `domain:bullfighting` is a `subClassOf` `domain:sport`, so a query to extract all sporting terms across all languages would identify the bullfighting term in this example. When combined with our bilingual dictionaries, which provide sense-level and domain-specific translations, or external, multilingual, public domain lexical resources, such an approach is potentially very powerful, as discussed in Section 5, "Next Steps".

Example 5. Spanish entry for abanicar in RDF/XML

```
<Lemma rdf:about="lemma:es_verb_abanicar">
  <rdfs:label xml:lang="es">abanicar</rdfs:label>
  <hasSource rdf:resource="source:oxford_gls_sudc_vox_dgle_exb1_5"/>
  <hasLanguage rdf:resource="lang:es"/>
  <hasPartOfSpeech rdf:resource="partOfSpeech:verb"/>
  <hasSense rdf:resource="sense:es_verb_abanicar_se_1"/>
  <hasSense rdf:resource="sense:es_verb_abanicar_se_2"/>
</Lemma>

<Sense rdf:about="sense:es_verb_abanicar_se_1">
  <isDescribedBy rdf:resource="definition:es_verb_abanicar_se_1_def_1"/>
  <hasExample rdf:resource="example:es_verb_abanicar_se_1_ex_1"/>
</Sense>

<Sense rdf:about="sense:es_verb_abanicar_se_2">
  <isDescribedBy rdf:resource="definition:es_verb_abanicar_se_2_def_1"/>
  <hasDomain rdf:resource="domain:bullfighting"/>
</Sense>

<StandardDefinition rdf:about="definition:es_verb_abanicar_se_1_def_1">
  <rdfs:label xml:lang="es">Dar aire con el abanico u otro objeto:</rdfs:label>
</StandardDefinition>

<StandardDefinition rdf:about="definition:es_verb_abanicar_se_2_def_1">
  <rdfs:label xml:lang="es">Incitar al toro agitando ante él el capote
  de un lado a otro, generalmente para que cambie de lugar en la
  suerte de varas.</rdfs:label>
</StandardDefinition>

<Example rdf:about="example:es_verb_abanicar_se_1_ex_1">
  <rdfs:label xml:lang="es">Pedro se abanica descuidadamente con su
  sombrero de copa y muestra un gesto de aburrimiento.</rdfs:label>
</Example>
```

5. Next Steps

At this stage, the output XML meets our current requirements, retaining both human readability and scope for further machine processing, and we can turn our attention to creating RDF. Because we provided for this during data modelling and developed the XML Schema and ontology in parallel, the XML readily converts into our RDF model. Transformation is as straightforward as writing some simple XSLT and adding a step to the XProc pipeline.

Once the build process generates manifold-dictionary RDF, we use that to populate a triplestore, which we can query for custom, multi-source, multilingual data-sets. For example, the following SPARQL query extracts English musical terms and their synonyms, along with their Spanish translations.

Example 6. SPARQL query to extract English and Spanish musical terms

```
SELECT ?txt ?tran ?syntxt ?syntran
WHERE {
  ?lemma rdfs:label ?txt ;
    :hasLanguage lang:en ;
    :hasSense ?sense .
  OPTIONAL {?lemma :hasEtymology ?et .
     ?et :originatesFrom ?etl ;
         :hasDateOfOrigin ?etd .}
  ?sense :hasSynonym ?syn ;
    :hasDomain [rdfs:subClassOf domain:music ] ;
    :hasTranslation ?t .
  ?t rdfs:label ?tran ;
     :hasLanguage lang:es .
  ?syn rdfs:label ?syntxt ;
    :hasSense ?synsense .
  ?synsense :hasTranslation ?synt ;
    :hasDomain [rdfs:subClassOf domain:music ] .
  ?synt rdfs:label ?syntran ;
        :hasLanguage lang:es .
}
```

In plain English, the query finds English lemmas that have a sense in the domain of music (or subclass thereof) that sense has a synonym and a Spanish translation. The synonym lemma must also have a sense in the domain of music as well as a Spanish translation. Using the SgVizler Javascript libraries for visualising SPARQL query results [13], we can create an interactive graph showing the terms coloured by date of origin:

Figure 4. Graph of musical terms in English and Spanish

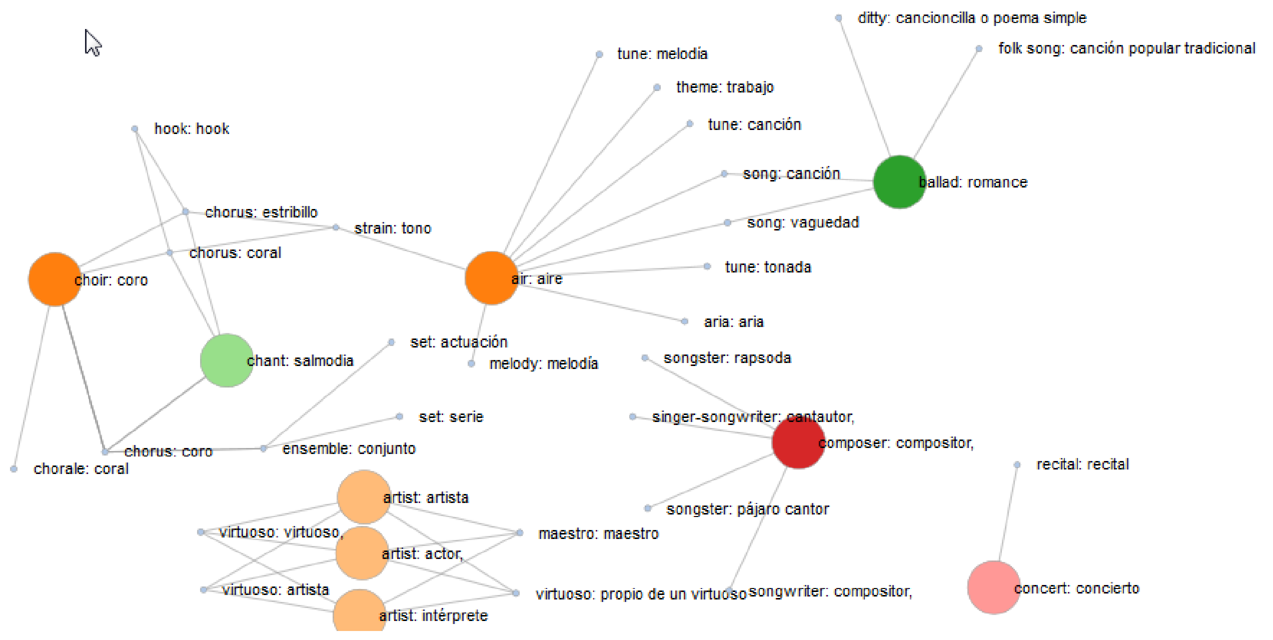
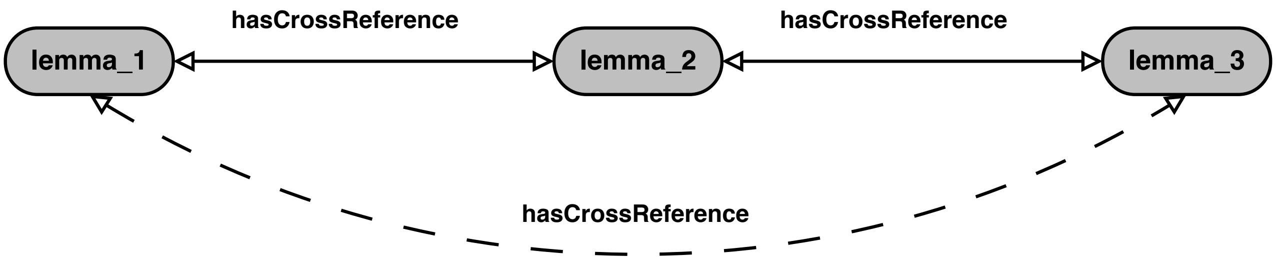


Figure 5. Inference via transitive property



Having the data in RDF and OWL also facilitates the discovery of new relationships which were not explicitly stated in the original XML data. This can be achieved through inference by using a reasoner or a rule language. For example, consider the transitive property depicted in Figure 5, "Inference via transitive property".
By modelling the property `hasCrossReference` using `owl:TransitiveProperty` a reasoner can infer a new relationship between lemma_1 and lemma_3 which was not present before.

More complex inference statements can be enforced using a rule language such as SWRL [14]. For example, a rule for inferencing new antonyms could be expressed in SWRL as follows (the rule has been simplified for sake of clarity):

```
Lemma(?x), Lemma(?y), hasAntonym(?x, ?y), hasSynonym(?y, ?z)
    -> hasAntonym(?x, ?z)
```

6. Conclusion

Our move towards a simpler, more consistent and harmonised XML data model for our dictionaries allows us to generate rich RDF representations of lexical data with a relatively straightforward transformation process. This is still very much an experimental approach, but it allows us to explore and visualise our lexical data across domains and languages in ways that were previously impossible when the content was held in separate silos in inconsistent XML representations.

Bibliography

[1] *The Oxford English Dictionary*. Oxford University Press. http://www.oed.com
[2] *Oxford Dictionaries*. Oxford University Press. http://www.oxforddictionaries.com
[3] *Oxford Global Language Solutions*. Oxford University Press. http://www.oup.com/ogls
[4] *Schematron*. ISO/IEC. http://www.schematron.com
[5] *Resource Description Framework*. World Wide Web Consortium. http://www.w3.org/standards/techs/rdf
[6] *OWL 2 Web Ontology Language*. World Wide Web Consortium. http://www.w3.org/TR/owl2-overview/
[7] *XProc: An XML Pipeline Language*. World Wide Web Consortium. http://www.w3.org/TR/xproc
[8] *XSpec - BDD Framework for XSLT*. Tennison, Jeni. http://code.google.com/p/xspec
[9] *Jenkins*. Jenkins CI. http://jenkins-ci.org
[10] *Apache Subversion*. Apache Software Foundation. http://subversion.apache.org
[11] *The Apache Ant Project*. Apache Software Foundation. http://ant.apache.org
[12] *CURIE Syntax 1.0: A syntax for expressing compact URIs*. World Wide Web Consortium http://www.w3.org/TR/curie
[13] *Sgvizler*. World Wide Web Consortium. http://www.w3.org/2001/sw/wiki/Sgvizler
[14] *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. World Wide Web Consortium. http://www.w3.org/Submission/SWRL

XML Processing in Scala

Dino Fancellu

Felstar Ltd

`<dino@felstar.com>`

William Narmontas

Apt Elements Ltd

`<william@scalawilliam.com>`

Abstract

Scala is an established static- and strongly-typed functional and object-oriented scalable programming language for the JVM with seamless Java interoperation.

Scala and its ecosystem are used at LinkedIn, Twitter, Morgan Stanley among many companies demanding remarkable time to market, robustness, high performance and scalability.

This paper shows you Scala's strong native XML support, powerful XQuery-like constructs, hybrid processing via XQuery for Scala, and increased XML processing performance. You will learn how you can benefit from Scala's practicality in a commercial setting, ultimately increasing your productivity.

Keywords: Scala, XML, XQuery, XSLT, XQJ, Java, Processing

1. Introduction

Programming style: Scala's immutability, functional programming, first-class XML make it rather similar to XQuery. Scala's for-comprehensions were inspired by Philip Wadler from his work with XQuery. [1]

Ecosystem: Scala's seamless Java interoperation gives you access to all of Java's libraries, the JVM [2] and many outstanding Scala libraries [1] [2] [3].

Scalability: Scala's scalability and design negate the need for design patterns in solving a language's design flaws. It is everything that Java should have been.

XML handling: Scala's XML handling includes the standard XML types such as `Element`, `Attribute`, `Node`. It also includes the `NodeSeq` type which extends `Seq[Node]` (a sequence of nodes), meaning that all of Scala's collections functionality for sequences is available for XML types. The key Scala XML documentation can be found at its author's Burak Emir's Scala XML book [3], scala.xml API [4] and scala-xml GitHub repository [5] .

2. Five minutes to understanding Scala

This paper covers a relevant selection of Scala's capabilities. There are many great resources to learn about traits, partial functions, case classes, etc. We will cover the necessary essentials for this paper. See Scala crash course [4] and a selected presentation [5] for detailed walk-throughs.

Like with XQuery and other functional programming languages we recommend programming Scala in an immutable fashion, although Scala allows you to program in an Object Oriented fashion or hybrid of the two, making it especially suited to migrating from a Java code base.

Scala's types are static, strong and mostly inferred, to the extent that it can feel like a scripting language [6] . Your IDE and Scala's compiler will inform you of your program's correctness very early on - including XML well-formedness.

Scala's 'implicits' enable you to define new methods on values in a limited scope. With implicits and type inference your code becomes very compact [7] [8]. In fact, this paper displays types only for the sake of clarity.

[1] ScalaTest - http://scalatest.org
[2] Akka - http://akka.io/
[3] Play Framework - http://www.playframework.com/
[4] scala.xml API - http://www.scala-lang.org/files/archive/nightly/docs/xml/
[5] scala-xml GitHub repository - https://github.com/scala/scala-xml/

doi:10.14337/XMLLondon14.Narmontas01

Scala is about expressions, not statements. The last expression in a block of expressions is the return value. The same applies to if-statements and try-catch.

Scala is best used from within IntelliJ IDEA and Eclipse with the Scala IDE plug-in. [9]

2.1. Values and functions

Scala & XQuery:

- `def fun(params): type` similar to
  `declare function local:fun(params): type`
- `val xyz = {expression}` similar to
  `let $xyz := {expression}`

Functions can be passed around easily. Example:

```
def incrementedByOne(x: Int) = x + 1

(1 to 5).map(incrementedByOne)
```

```
Vector(2, 3, 4, 5, 6)
```

This example however can be slimmed down to

```
(1 to 5).map(x => x + 1)
```

```
Vector(2, 3, 4, 5, 6)
```

Where `x => x + 1` is an anonymous (lambda) function. It can be slimmed down further to

```
(1 to 5).map(_+1)
```

```
Vector(2, 3, 4, 5, 6)
```

Scala's collections, such as lists, sets and maps come in mutable and immutable flavours [10] . They will be used throughout the examples.

2.2. Strings and string interpolation

The triple double-quote syntax negates escaping of double-quotes in string literals. E.g.

```
val title = """An introduction to "Scala""""
```

Scala supports string interpolation [11] similar to that in PHP, Perl and CoffeeScript - with the 's' modifier:

```
val language = "Scala"
val interpolatedTitle =
  s"""An introduction to "$language""""
```

String interpolation turns `$language` into `${language.toString}`.

Scala's triple-quoted strings may be multi-line, as shown in the examples section.

2.3. Named parameters

Where further clarity for method calls is needed, you can use named parameters:

```
def makeLink(url: String, text: String) =
  s"""<a href="$url">$text</a>"""

makeLink(text = "XML London 2014",
  url="http://www.xmllondon.com/")
```

```
<a href="http://www.xmllondon.com/">
 XML London 2014</a>
```

2.4. For-comprehensions

For-comprehensions [12] will be familiar to a programmer who has used Python, LINQ, XQuery, Ruby, Haskell, F#, Erlang, Clojure.

You can rewrite the previous example `(1 to 5).map(x => x + 1)` as a for-comprehension:

```
for ( x <- (1 to 5) ) yield x + 1
```

```
Vector(2, 3, 4, 5, 6)
```

These comprehensions `yield` results by iterating over multiple collections:

```
val software = Map(
  "Browser" -> Set("Firefox", "Chrome",
    "Internet Explorer"),
  "Office Suite" -> Set(
    "Google Drive", "Microsoft Office",
    "Libre Office")
)
for { (softwareKind, programs) <- software
       program <- programs
       if program endsWith "e"
} yield s"$softwareKind: $program"
```

```
List(Browser: Chrome, Office Suite: Google Drive,
 Office Suite: Microsoft Office,
 Office Suite: Libre Office)
```

Inside a for-comprehension, Scala and XQuery once again share similarities:

- `x <- {expression}` similar to
  `for $x in {expression}`
- `if {condition}` similar to
  `where {condition}`
- `abc = {expression}` similar to
  `let $abc := {expression}`
- `yield {expression}` similar to
  `return {expression}`

3. Scala's strong native XML support

Unlike in Java, XML is a first class citizen in Scala and can be used as a native data type.

The scala.xml library source code is available on GitHub.[1]

3.1. Basic Inline XML

XML literals can be embedded directly in code with curly braces.

```
val title = "XML London 2014"
val xmlTree = <div>
  <p>Welcome to <em>{title}</em>!</p>
</div>
```

Serializing this XML structure works as expected:

```
xmlTree.toString
```

```
<div>
  <p>Welcome to <em>XML London 2014</em>!</p>
</div>
```

These XML literals are checked for well formedness at compile time or even in your IDE reducing errors.

Curly braces can be escaped with double braces. e.g.

```
val squiggles = <root>I like {{squiggles}}</root>
```

```
<root>I like {squiggles}</root>
```

3.2. Reading

Scala can load XML from Java’s `File`, `InputStream`, `Reader`, `String` using the `scala.xml.XML` object. Here is an XML document in `String` form:

```
val pun =
"""<pun rating="extreme">
|  <question>Why do CompSci students need
|glasses?</question>
|  <answer>To C#<!--
|C# is a Microsoft's programming language
|-->.</answer>
|</pun>""".stripMargin
```

Loading an XML document from a String gives us a node:

```
scala.xml.XML.loadString(pun)
```

```
<pun rating="extreme">
  <question>Why do CompSci students need
glasses?</question>
  <answer>To C#.</answer>
</pun>
```

When you need XML comments use the `ConstructingParser` [13] :

```
scala.xml.parsing.ConstructingParser
.fromSource(scala.io.Source.fromString(pun),
preserveWS = true).document().docElem
```

```
<pun rating="extreme">
  <question>Why do CompSci students need
glasses?</question>
  <answer>To C#<!--
C# is a Microsoft's programming language
-->.</answer>
</pun>
```

3.2.1. Look ups and XPath alternatives

Scala has its own XPath-like methods for querying from XML trees

```
val listOfPeople = <people>
  <person>Fred</person>
  <person>Ron</person>
  <person>Nigel</person>
</people>
listOfPeople \ "person"
```

```
NodeSeq(<person>Fred</person>,
 <person>Ron</person>, <person>Nigel</person>)
```

Wildcard is similar

```
listOfPeople \ "_"
```

```
NodeSeq(<person>Fred</person>,
 <person>Ron</person>, <person>Nigel</person>)
```

Looking for descendants

```
val fact = <fact type="universal">
<variable>A</variable> = <variable>A</variable>
</fact>
fact \\ "variable"
```

```
NodeSeq(<variable>A</variable>,
 <variable>A</variable>)
```

Querying attributes is similar

```
fact \ "@type"
```

```
: scala.xml.NodeSeq = universal
```

```
fact \@ "type"
```

```
: String = universal
```

[1] scala-xml library GitHub - https://github.com/scala/scala-xml/

Looking up elements by namespace (see Appendix A, *The showNamespace(-s) methods* for `showNamespaces`):

```
val tree = <document>
  <embedded xmlns="urn:test:embedding">
    <description>
      <referenced xmlns="urn:test:referencing">
        <metadata>
          <title xmlns="">Untitled</title>
        </metadata>
      </referenced>
    </description>
  </embedded>
</document>

(tree \\ "_").
  filter(_.namespace == "urn:test:referencing").
  map(showNamespace).foreach(println)
```

```
{urn:test:referencing}referenced
{urn:test:referencing}metadata
```

Looking up attributes by namespace:

```
<node xmlns="urn:meta" demo="test"/> \ "@demo"
```

```
test
```

```
<node xmlns:meta="urn:meta" meta:demo="test"/> \
  "@{urn:meta}demo"
```

```
test
```

The reason that backslashes were chosen instead of the usual forward slashes is due to the use of // for Scala comments. i.e. the // would never even be seen.
Scala's XML is displayed as a NodeSeq type which extends `Seq[Node]`. This means we get Scala's collections for free. Here are some examples:

```
val root = <numbers>
  {for {i <- 1 to 10} yield
    <number>{i}</number>}
</numbers>
val numbers = root \ "number"
numbers(0)
```

```
<number>1</number>
```

```
numbers.head
```

```
<number>1</number>
```

```
numbers.last
```

```
<number>10</number>
```

```
numbers take 3
```

```
NodeSeq(<number>1</number>, <number>2</number>,
 <number>3</number>)
```

```
numbers filter(_.text.toInt > 6)
```

```
NodeSeq(<number>7</number>, <number>8</number>,
 <number>9</number>, <number>10</number>)
```

The default `apply` method for `NodeSeq` is an alias for `filter`:

```
numbers(_.text.toInt > 6)
```

```
NodeSeq(<number>7</number>, <number>8</number>,
 <number>9</number>, <number>10</number>)
```

```
numbers maxBy(_.text)
```

```
<number>9</number>
```

```
numbers maxBy(_.text.toInt)
```

```
<number>10</number>
```

```
numbers.reverse
```

```
NodeSeq(<number>10</number>, <number>9</number>,
 <number>8</number>, <number>7</number>,
 <number>6</number>, <number>5</number>,
 <number>4</number>, <number>3</number>,
 <number>2</number>, <number>1</number>)
```

```
numbers.groupBy(_.text.toInt % 3)
```

```
Map(
 2 -> NodeSeq(<number>2</number>,
 <number>5</number>, <number>8</number>),
 1 -> NodeSeq(<number>1</number>,
 <number>4</number>, <number>7</number>,
 <number>10</number>),
 0 -> NodeSeq(<number>3</number>,
 <number>6</number>, <number>9</number>))
```

```
val jokes = <jokes>
  <pun rating="fine">
    <question>Q: Why did the functions stop
calling each other?</question>
    <answer>A: Because they had constant
arguments.</answer>
  </pun>
  <pun rating="extreme">
    <question>Why do
CompSci students need glasses?</question>
    <answer>To C#<!--
C# is a Microsoft programming language
-->.</answer>
  </pun>
</jokes>
```

Querying descendant attributes works as expected

```
jokes \\ "@rating"
```

```
NodeSeq(fine, extreme)
```

Querying elements by path works fine

```
jokes \ "pun" \ "question"
```

```
NodeSeq(<question>Q: Why did the functions stop
calling each other?</question>, <question>Why do
CompSci students need glasses?</question>)
```

Querying attributes by path:

```
jokes \ "pun" flatMap (_\ "@rating")
```

```
NodeSeq(fine, extreme)
```

```
(jokes \ "pun") \\ "@rating"
```

```
NodeSeq(fine, extreme)
```

However node equality can surprise with XML literals [1]:

```
<node>{2}</node> == <node>2</node>
```

```
false
```

```
<node>{2}</node> == <node>{2}</node>
```

```
true
```

3.3. Scala XML namespace handling

Namespaces are handled well. The empty namespace is 'null'. (see Appendix A, *The showNamespace(-s) methods* for showNamespaces):

```
val tree = <document>
  <embedded xmlns="urn:test:embedding">
    <description>
      <referenced xmlns="urn:test:referencing">
        <metadata>
          <title xmlns="">Untitled</title>
        </metadata>
      </referenced>
    </description>
  </embedded>
</document>

showNamespaces(tree)
```

```
{null}document
{urn:test:embedding}embedded
{urn:test:embedding}description
{urn:test:referencing}referenced
{urn:test:referencing}metadata
{null}title
```

3.3.1. Scala XML is unidirectional and immutable

Unlike the XPath model, Scala XML is unidirectional, i.e. a node does not know its parent, so lacks reverse axes, also no forward/sibling axes. This was done because adding in parents is expensive whilst maintaining immutability. For many problem spaces that may not matter. If it does for you then you are free to fall back to the full XPath/XQuery/XSLT model as shown below

3.3.2. XQS

We use a tiny wrapper library called XQS (XQuery for Scala) [14] in various places throughout this paper. Its main aim is to allow for a Scala metaphors when using XQuery. However even outside of XQuery usage, it allows for easy interoperation between the worlds of Scala XML and Java DOM. For example in the XPath example below, it supplies toDom to turn Scala XML to a w3c DOM, and the ability to turn a NodeSet into a Scala NodeSeq.

3.3.3. Using XPath from Scala

```
import com.felstar.xqs.XQS._
val widgets = <widgets>
  <widget>Menu</widget>
  <widget>Status bar</widget>
  <widget id="panel-1">Panel</widget>
  <widget id="panel-2">Panel</widget>
</widgets>
val xpath = XPathFactory.newInstance().newXPath()
val nodes: NodeSeq = xpath.evaluate(
  "/widgets/widget[not(@id)]",
  toDom(widgets),
  XPathConstants.NODESET
).asInstanceOf[NodeList]
nodes
```

```
NodeSeq(<widget>Menu</widget>,
 <widget>Status bar</widget>)
```

Natively in Scala:

```
(widgets \ "widget")(
  widget => (widget \ "@id").isEmpty
)
```

```
NodeSeq(<widget>Menu</widget>,
 <widget>Status bar</widget>)
```

[1] https://github.com/scala/scala-xml/issues/25

3.3.4. XML Transformations

Scala provides XML transformation functionality via a `RuleTransformer` that takes multiple `RewriteRules`. The following example uses pattern matching and a native XML extractor:

```
val peopleXml = <people>
    <john>Hello, John.</john>
    <smith>Smith is here.</smith>
    <another>Hello.</another>
  </people>

val rewrite =
  new RuleTransformer(new RewriteRule {
    override def transform(node: Node) =
      node match {
        case <john>{_}</john> =>
          <john>Hello, John.</john>
        case <smith>{text}</smith> =>
          <smithX>{text}!!!!</smithX>
        case n: Elem if n.label != "people" =>
          n.copy(label = "renamed")
        case other => other
  }
})
rewrite.transform(peopleXml)
```

```
<people>
    <john>Hello, John.</john>
    <smithX>Smith is here.!!!!</smithX>
    <renamed>Hello.</renamed>
  </people>
```

Alternatively: calling XSLT from Scala

```
val stylesheet =
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">
  <xsl:template match="john">
    <xsl:copy>Hello, John.</xsl:copy>
  </xsl:template>
  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
import com.felstar.xqs.XQS._
val xmlResultResource = new java.io.StringWriter()
val xmlTransformer =
  TransformerFactory
  .newInstance().newTransformer(stylesheet)
xmlTransformer.transform(peopleXml,
  new StreamResult(xmlResultResource))
xmlResultResource.getBuffer
```

```
<?xml version="1.0" encoding="UTF-8"?><people>
    <john>Hello, John.</john>
    <smith>Smith is here.</smith>
    <another>Hello.</another>
  </people>
```

We found XSLT more effective than Scala for designing XML transformations as XSLT has been designed explicitly for this task. Thus we can mix-and-match transformations when XSLT is nicer than Scala and vice-versa. John Snelson's transform.xq showcases mixed transforms with querying in XQuery [15]. Alike can be achieved in Scala.

3.3.5. XML Pull Parsing from Scala

```
// 4GB file, comes back in a second.
val downloadUrl =
  "http://dumps.wikimedia.org" +
  "/enwiki/20140402/enwiki-20140402-abstract.xml"
val src = Source.fromURL(downloadUrl)
val er = XMLInputFactory.newInstance().
  createXMLEventReader(src.reader)

implicit class XMLEventIterator(ev: XMLEventReader)
  extends scala.collection.Iterator[XMLEvent] {
  def hasNext = ev.hasNext
  def next = ev.nextEvent()
}

er.dropWhile(!_.isStartElement).take(10)
  .zipWithIndex.foreach {
    case (ev, idx) =>
      println(s"${idx+1}:\t$ev")
  }

src.close()
```

```
1:  <feed>
2:

3:  <doc>
4:

5:  <title>
6:  Wikipedia: Anarchism
7:  </title>
8:

9:  <url>
10: http://en.wikipedia.org/wiki/Anarchism
```

3.3.6. Calling XQuery from Scala

The standard API for XQuery on Java is XQJ [16]. XQJ drivers are available for several databases such as MarkLogic and XQuery processors such as Saxon [17] [18] meaning Scala can consume XQuery result sets.

```
import com.felstar.xqs.XQS._
val conn = getYourXQueryConnection()
val ret: NodeSeq = conn(
  "/widgets/widget[not(@id)]", widgets)
```

```
NodeSeq(<widget>Menu</widget>,
 <widget>Status bar</widget>)
```

```
val ret2: NodeSeq = conn(
  """|for $w in /widgets/widget
     |order by $w return $w""".stripMargin,
  widgets)
```

```
NodeSeq(<widget>Menu</widget>,
 <widget id="panel-1">Panel</widget>,
 <widget id="panel-2">Panel</widget>,
 <widget>Status bar</widget>)
```

A pure Scala version:

```
(widgets \ "widget").sortBy(_.text)
```

4. Extensibility

Using Scala's "implicits" you can enrich types by adding new functionality.

```
val oo = <oo>
<x id="1">123</x>
<x id="2">1234</x>
<x id="x">xxxxx</x>
<x id="3">1235</x>
</oo>
```

Treating attribute values, which are strings, as doubles, implicitly when needed, and without any NumberFormatExceptions. Uses the scala.util.Try class that wraps exceptions in a functional manner

```
implicit def toSafeDouble(st: String) =
  scala.util.Try{st.toDouble}.getOrElse(Double.NaN)

(oo \ "x").filter( _ \@ "id" < 3)
```

```
NodeSeq(<x id="1">123</x>, <x id="2">1234</x>)
```

```
(oo \\ "@id").map(_.text: Double).
  filterNot(_.isNaN).sum
```

```
6.0
```

Here are some examples of selecting multiple items according to their index:

```
val root = <nodes>
  <node>a (0)</node>
  <node>b (1)</node>
  <node>c (2)</node>
  <node>d (3)</node>
  <node>e (4)</node>
  <node>f (5)</node>
  <node>g (6)</node>
  <node>h (7)</node>
  <node>i (8)</node>
</nodes>
val nodes = (root \ "node")

implicit class indexFunctionality(ns: NodeSeq) {
  def filterByIndex(p: Int => Boolean): NodeSeq =
    ns.zipWithIndex.collect {
      case (value, index) if p(index) => value
    }
  def filterByIndex(b: GenSeq[Int]*): NodeSeq=
    filterByIndex(b.flatten.toSet)
  def apply(n1: Int, n2: Int*) =
    ns(n1) ++ ns.filterByIndex(n2.toSet)
  def apply(b: GenSeq[Int]*): NodeSeq =
    filterByIndex(b.flatten.toSet)
}
nodes.filterByIndex(_ > 6)
```

```
NodeSeq(<node>h (7)</node>, <node>i (8)</node>)
```

```
nodes(0, 4, 7)
```

```
NodeSeq(<node>a (0)</node>, <node>e (4)</node>,
 <node>h (7)</node>)
```

```
nodes(1 to 3)
```

```
NodeSeq(<node>b (1)</node>, <node>c (2)</node>,
 <node>d (3)</node>)
```

```
nodes(1 to 3, 5 until 7)
```

```
NodeSeq(<node>b (1)</node>, <node>c (2)</node>,
 <node>d (3)</node>,
 <node>f (5)</node>, <node>g (6)</node>)
```

Note that root can alternatively be generated using:

```
val root = <nodes> {('a' to 'i').zipWithIndex.map{
  case (letter, index) =>
    <node>{letter} ({index})</node>
}}</nodes>
```

This is how we lookup elements by namespace. You can see how extensible Scala becomes (using Appendix A, *The showNamespace(-s) methods*):

```
val tree = <document>
  <embedded xmlns="urn:test:embedding">
    <description>
      <referenced xmlns="urn:test:referencing">
        <metadata>
          <title xmlns="">Untitled</title>
        </metadata>
      </referenced>
    </description>
  </embedded>
</document>
implicit class nsElement(nodeSeq: NodeSeq) {
  val regex = """^\{(.+)\}(.+)$""".r
  def \\#(path: String): NodeSeq = {
    val regex(namespace, el) = path
    for {
      node <- nodeSeq \\ el
      if node.namespace == namespace
    } yield node
  }
  def \#(path: String): NodeSeq = {
    val regex(namespace, el) = path
    for {
      node <- nodeSeq \ el
      if node.namespace == namespace
    } yield node
  }
}

(tree \\# "{urn:test:referencing}_").
  map(showNamespace).mkString("\n")
```

```
{urn:test:referencing}referenced
{urn:test:referencing}metadata
```

4.1. Further Extensibility: XQuery-like constructs

Here we implement the XQuery 3.0 use case Q4 Part 3 [19].

XQuery code:

```
<result>{
  for $store in /root/*/store
  let $state := $store/state
  group by $state
  order by $state
  return
    <state name="{$state}">{
      for $product in /root/*/product
      let $category := $product/category
      group by $category
      order by $category
      return
        <category name="{$category}">{
          for $sales in /root/*/record[
            store-number = $store/store-number
            and product-name = $product/name]
          let $pname := $sales/product-name
          group by $pname
          order by $pname
          return
            <product name="{$pname}"
            total-qty="{sum($sales/qty)}"/>
          }</category>
    }</state>
}</result>
```

Scala code:

```
def loadXML(ref: String) = {
  val filename = s"benchmarks-xml/$ref"
  val file = new File(filename)
  scala.xml.XML.loadFile(file)
}

val allStores = loadXML("stores.xml") \ "store" groupByOrderBy "state"
val allProducts = loadXML("products.xml") \ "product" groupByOrderBy "category"
val allRecords = loadXML("sales-records.xml") \ "record" groupByText "product-name"

<result>{
  for {
    (state, stateStores) <- allStores
    storeNumbers = (stateStores \ "store-number").textSet
  } yield <state name={state}>{
    for {
      (category,products)<- allProducts
      productRecords = allRecords.filterKeys{(products\"name").textSet}
    } yield <category name={category}>{
      for {
        (productName, productSales)<- productRecords
        filteredSales = productSales.filter(n => storeNumbers(n\"store-number" text) )
        if filteredSales.nonEmpty
        totalQty = (filteredSales \ "qty").map(_.text.toInt).sum
      }
      yield <product name={productName} total-qty={totalQty.toString}/>
      }</category>
    }</state>
  }</result>
```

An extensibility class used is attached in Appendix B, *Extensions for NodeSeq*.

5. Performance vs XQuery

5.1. Assumptions

Core i7-3820 @ 3.6 GHz, 4 core, Windows 7 Professional, 64 bit, 16 GB Ram, Java 7 u51 64bit, default JVM settings. Scala 2.11, XMLUnit, XQJ interfaces, XQS Scala bindings 2 XQuery implementations A and B. Sources are located on GitHub [20].

5.2. Methodology

Using prepared statements for the XQuery, can be switched off, B performance drops like a stone without it, and not really fair, so turn on prepared statements. Scala has no concept of these, as there is nothing to prepare or parse. Also cached the conversion of XML to a DOMSource for the XQuery, so we don't measure that effort when timing the XQueries. Put in switch to serialize results to string, so as to ensure that any potential lazy values are materialized. Selected various queries from [21] also a XQuery 3.0 example from [19]. Runs both XQuery and Scala in a single run, 3 runs of 10,000 queries, with the results of the first 2 runs thrown away to get a good JVM jit warmup. Its very easy to get misleading results from badly thought out benchmarks. Warm up is very important, JVM runs best when code is hotspotted. For each query we emits the XQuery time, Scala time, and the ratio of these times, XQuery:Scala. We plot a graph of these values, showing first 2 values as a bar, the ratio as a line.

5.3. Benchmarks

Impl A XQuery vs Scala (Prep Statements, serialized)

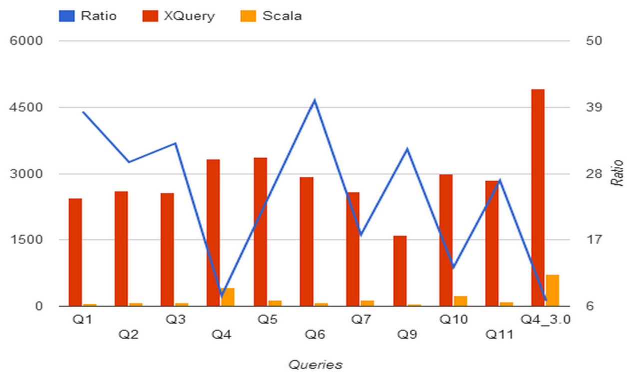


Table 1. Impl A XQuery vs Scala

Query	Ratio	XQuery	Scala
Q1	38.27	2449	64
Q2	29.89	2600	87
Q3	33	2574	78
Q4	7.75	3325	429
Q5	23.91	3372	141
Q6	40.1	2927	73
Q7	17.86	2590	145
Q9	32.04	1602	50
Q10	12.48	2994	240
Q11	26.86	2847	106
Q4_3.0	6.89	4921	714

Impl B XQuery vs Scala (Prep Statements, serialized)

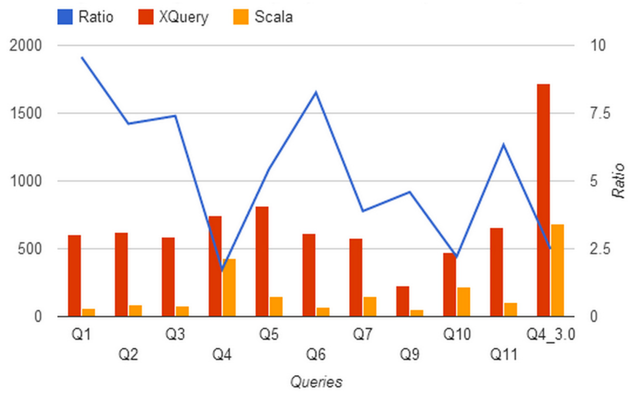


Table 2. Impl B XQuery vs Scala

Query	Ratio	XQuery	Scala
Q1	9.57	603	63
Q2	7.11	619	87
Q3	7.4	592	80
Q4	1.74	750	430
Q5	5.44	816	150
Q6	8.26	611	74
Q7	3.89	579	149
Q9	4.59	225	49
Q10	2.21	478	216
Q11	6.33	658	104
Q4_3.0	2.49	1715	688

5.4. Conclusions

Scala is faster in all these use cases. Very similar to XQuery in its language construction. No doubt there are use cases where XQuery may be better, like an XML database. This is not black or white, a religious issue, simply a matter of choice.

6. Practicality

6.1. Enterprise usage

Scala is well established in enterprises [22]. While having access to the JVM Scala makes it easy to reuse the solid and tested libraries of the JVM ecosystem as well as an enterprise's legacy Java code [23]. Scala's terseness makes domain modelling much more precise [24]. Enterprise can migrate slowly to using all-Scala. The amount of code to maintain decreases, so number of moving parts decreases.

6.2. ScalaTest

ScalaTest, then test either your whole domain with property based testing, and ensure that the parties you are dealing with understand what your XML processing code does. Again, whether your XML processing code is inside Scala, XSLT, XQuery or MarkLogic, makes no difference. XMLUnit works nicely with Scala.

6.3. Other integration features

Scala 2.11 makes itself available as a scripted language to JSR-223 [25]. Scala's Akka [1] and [2] provide many integration features with the rest of the world including JSON [3] and WebSockets [4]. With macros you can create programs that create programs. Meaning your language is not getting in your way with 'design patterns' when focusing on the problem you're trying to solve. This includes creating bindings such as serializers and deserializers of your favourite formats (e.g. binary via Scala Pickling [5], JSON via json4s [6].

We would like to see more research in querying with Scala such as Fatemeh Borran-Dejnabadi's paper [26] .

7. Conclusions

Possibilities with using Scala for XML processing are almost limitless. Pick and mix how you want to process your XML in Scala: powerful collections methods, for-comprehensions, XML generation, XPath, XSLT, XML databases and XQuery engines via XQS/XQJ, XML streaming via StAX. Scala makes it possible to simplify complex logic into domain specific programs and use a combination of the best tools for achieving your targets. As Java has not advanced as far in terms of the language, Scala has secured the niche of the effective programmer and the effective business. For you as a functional programmer Scala's concepts will already be familiar. You lose none of your existing Java ecosystem and gain so much more. It is another important tool in your armoury for efficient and lucid data processing.

[1] Akka - http://akka.io/
[2] Play framework - http://www.playframework.com/
[3] http://www.playframework.com/documentation/2.2.x/ScalaJson
[4] Play Framework documentation, WebSockets in Scala guide - http://www.playframework.com/documentation/2.2.x/ScalaWebSockets
[5] http://lampwww.epfl.ch/~hmiller/pickling/
[6] http://json4s.org/

Bibliography

[1] *Martin Odersky on the Future of Scala (25:00)*. http://www.infoq.com/interviews/martin-odersky-scala-future Sadek Drobi and Martin Odersky. InfoQ.

[2] *What is Scala? Seamless Java interop*. Martin Odersky. http://www.scala-lang.org/what-is-scala.html#seamless_java_interop

[3] *Scala XML Book*. Burak Emir. https://sites.google.com/site/burakemir/scalaxbook.docbk.html?attredirects=0

[4] *Scala Crash Course*. February 20, 2014. University of California, San Diego. Ravi Chugh. http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/00-crash.html

[5] *Scala - The Short Introduction*. Jerzy Müller. http://scalacamp.pl/intro/#/start

[6] *Scala: The Static Language that Feels Dynamic*. Bruce Eckel. Artima, Inc.. June 12, 2011. http://www.artima.com/weblogs/viewpost.jsp?thread=328540

[7] *Implicit classes overview, Scala documentation*. http://docs.scala-lang.org/overviews/core/implicit-classes.html

[8] *Pimp my Library*. Martin Odersky. Artima, Inc.. October 9, 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=179766

[9] *Scala: Which is the best IDE for Scala Development?*. Quora. Navad Samet. January 13, 2014. http://www.quora.com/Scala/Which-is-the-best-IDE-for-Scala-Development/answer/Nadav-Samet-1

[10] *Scala Collections overview*. scala-lang.org. http://docs.scala-lang.org/overviews/collections/overview.html

[11] *String Interpolation*. scala-lang.org. Josh Suereth. http://docs.scala-lang.org/overviews/core/string-interpolation.html

[12] *Iteration & Recursion - Scala crash course*. Ravi Chugh. University of California, San Diego. February 27, 2014. http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/01-iterators.slides.html

[13] *scala.xml.parsing.ConstructingParser*. scala-lang.org. http://www.scala-lang.org/files/archive/nightly/docs/xml/#scala.xml.parsing.ConstructingParser

[14] *XQuery for Scala*. Dino Fancellu. https://github.com/fancellu/xqs

[15] *Transform.xq: A transformation library for XQuery 3.0*. John Snelson. XML Prague 2012. http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[16] *JSR 225: XQuery API for Java (XQJ)*. Maxim Orgiyan and Marc Van Cappellen. https://jcp.org/en/jsr/detail?id=225

[17] *XQJ.NET*. Charles Foster. http://xqj.net/

[18] *XQuery API for Java*. http://en.wikipedia.org/wiki/XQuery_API_for_Java.

[19] *XQuery 3.0 Use Cases - Group By Q4*. W3C Working Group. http://www.w3.org/TR/xquery-30-use-cases/#groupby_q4

[20] *Benchmark sources*. https://github.com/ScalaWilliam/XMLLondon2014/. Dino Fancellu.

[21] *XML Query Use Cases*. W3C Working Group. March 23, 2007. http://www.w3.org/TR/xquery-use-cases/

[22] *Case Studies & Stories*. Typesafe, Inc.. https://typesafe.com/company/casestudies

[23] *The Guardian case study*. Typesafe, Inc.. http://downloads.typesafe.com/website/casestudies/The-Guardian-Case-Study-v1.1.pdf

[24] *Implementing a DSL for Social Modeling: an Embedded Approach Using Scala*. Jesús López González. Juan Manuel. October 13, 2013. http://www.infoq.com/presentations/speech-dsl-social-process

[25] *SI-874 JSR-223 compliance for the interpreter*. https://github.com/scala/scala/pull/2238. Adriaan Moors.

[26] *Efficient Semi-structured Queries in Scala using XQuery Shipping*. Fatemeh Borran-Dejnabadi. February 2006. http://infoscience.epfl.ch/record/85493/files/Scala_XQuery.pdf

A. The showNamespace(-s) methods

```scala
def showNamespace(node: Node) =
  s"{${node.namespace}}${node.label}"

def showNamespaces(ofTree: NodeSeq) =
  (ofTree \\ "_").map(showNamespace).mkString("\n")
```

B. Extensions for NodeSeq

```scala
val XQueryEquals = new Equiv[NodeSeq] {
  def equiv(a: NodeSeq, b: NodeSeq): Boolean = {
    (a.map(_.text).toSet &
     b.map(_.text).toSet).size > 0
  }
}

implicit class nsExtensions(nodeSeq: NodeSeq) {

  def ===(b: NodeSeq): Boolean =
    XQueryEquals.equiv(nodeSeq, b)

  def ===(b: GenTraversable[String]): Boolean =
    b.exists(text =>
      XQueryEquals.equiv(nodeSeq, Text(text)))

  def ===(hasValue: String): Boolean =
    nodeSeq.text == hasValue

  def sortedByText: NodeSeq =
    nodeSeq.sortBy(_.text)

  def sortedText: Seq[String] =
    nodeSeq.map(_.text).sorted

  def sortedByLookup(ofPath: String): NodeSeq =
    nodeSeq.sortBy(node => (node \ ofPath).text)

  def groupByText(ofPath: String)
    : Map[String, NodeSeq] =
    nodeSeq.groupBy(node => (node \ ofPath).text)

  def groupByOrderBy(ofPath: String)
    : List[(String, NodeSeq)] =
    groupByText(ofPath).toList.sortBy(_._1)

  def textSet: Set[String] =
    nodeSeq.map(_.text).toSet

}
```

XML Authoring On Mobile Devices

George Bina

Syncro Soft / oXygen XML Editor

`<george@oxygenxml.com>`

Abstract

Not too long ago XML-born content was not present in a mobile-friendly form on mobile devices. Now, many of the XML frameworks like DocBook, DITA and TEI provide output formats that are tuned to be used on mobile devices. These are either different electronic book formats (EPUB, Kindle) or different mobile-friendly web formats.

Many people find XML authoring difficult on computers, let alone mobile devices. However, due to the constantly increasing number of mobile devices, that made people create mobile-friendly output formats from XML documents, there is clearly a need to provide also direct access to authoring XML content on these devices.

I would like to explore the options for providing XML authoring on mobile devices and describe our current work and the technology choices we made to create an authoring solution for mobile devices. Trying to enable people to create XML documents on mobile devices is a very exciting, mainly because the user interaction is completely different on a mobile device: different screen resolutions, different interaction methods (touch, swipe, pinch), etc. See how we imagined XML authoring on an Android phone or on iPad! How about editing XML on a smart TV? Leverage speech recognition/dictation and handwriting recognition technologies that are available on mobile devices to enable completely new ways of interacting with XML documents!

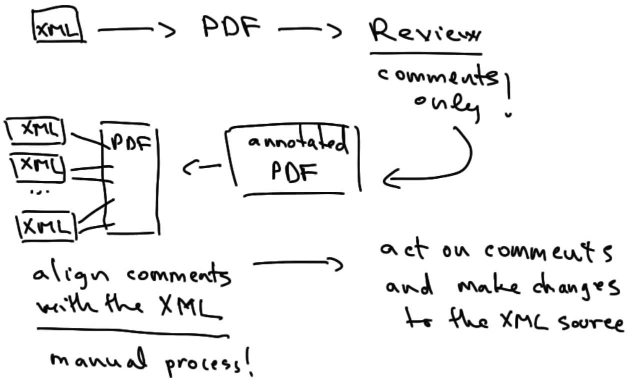
Keywords: XML, authoring, mobile, review, user experience

1. Introduction

When an XML-based solution is implemented the lowest impedance for communicating between different processing steps is to use XML, so people try to use XML in as many places as possible, but there are usually a few processes where using XML is not always easy. One of these processes is the review of XML content. Another process is the contribution of initial content from people that are not familiar with XML.

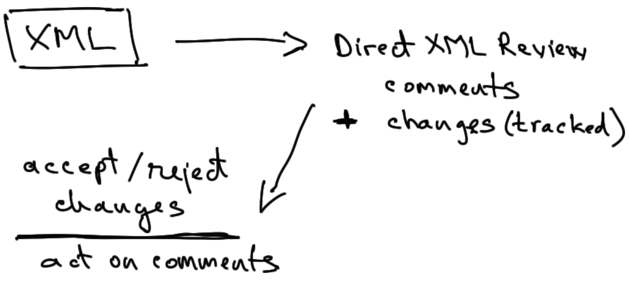
A traditional review process will convert the XML information to an output format, usually PDF, and have reviewers annotate that PDF with comments, then align the PDF with the XML documents that generated it to identify the places in the XML source the comments refer to and manually act on those comments to make the corresponding changes to the XML documents.

Figure 1. Traditional review process


XML
PDF
Review
comments only!
XML
XML
...
XML
PDF
annotated PDF
align comments with the XML
manual process!
act on comments and make changes to the XML source

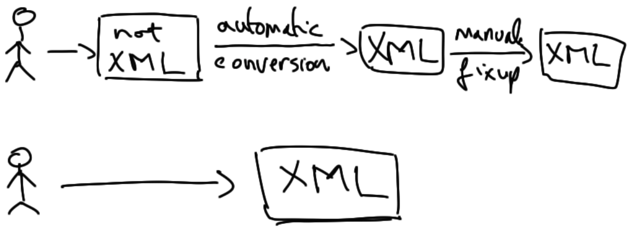
This process has many steps and some of them are not automated so they not only consume time but errors can appear at different stages. Many of the issues can be solved by adopting a direct XML review process, where users can annotate directly on the XML content, and add not only comments but also make changes to the document that will be considered proposed changes. Thus, responding to a comment by identifying the XML source the comment refers to and then updating the document as described in the comment can be replaced with an simple action to accept a proposed change to the document.

Figure 2. Direct XML review process



Contributing initial content is very much linked to the tools the users already know and the devices he has access to. Thus initial content is contributed in whatever format the users can use and then converted to XML to be able to enter the XML-based solution. Usually people use Word and there is a conversion process that tries to get from Word to XML. We can cut the conversion cost if we are able to get this initial content in XML form.

Figure 3. Non XML data conversion to XML vs XML first



When you move to an XML-based solution it is important to be able to cut costs on the review process and to implement an XML-first system, where people can contribute initial data directly in XML. We tried to address these problems and we currently provide solutions for both the review process and for creating an XML-first solution. However, the current solution requires the use of a laptop or a desktop computer.

The people that perform reviews or the ones that contribute initial content are in general external to the department that deals with the XML-based solution, so it is difficult to control the resources available to these people. The increasing use of mobile devices during the last years and the projections for next years show that mobile devices are not something we can ignore (mobile devices are expected to exceed the number of desktops this year) and the only device some of those people have may be a mobile one. So, if we want to be able to cut the costs and the complexity of processes similar to the ones described, we need to be able to provide at least direct XML review and simplified XML authoring on mobile devices.

2. Technology choices

Once we decided to start building a tool for XML authoring on mobile devices the next step was to decide on what technologies that will be based on. The first decision was if it was to be a native application or a web application.

From our experience with oXygen we found that it is great to be able to support multiple platforms with the same code - oXygen being built in Java works on any platform that provides a JVM. So one problem with a native solution was that we had to build a different application for each mobile platform while a web application will allow us to reuse the same code for all devices, as long as they support the required web technology (HTML5 and JavaScript). A web application has also other advantages over a native application like immediate update, no app-store interference and an important one - the fact that it will work also on desktops. A disadvantage will be that the access to device specific functionality will not be possible but it should be possible to use a hybrid application if such functionality will be critical in the future.

Another decision point was on how much processing should be done on the client and how much on the server. Targeting mobile devices we wanted to have as little as possible processing on the client, in order not to drain the device battery. This factor and the fact that we already have in Java many of the components needed for XML authoring made us decide to prefer the server side processing to the client processing and keep the current oXygen on the server and have only the display part on the client side, like a remote display, thus reusing almost all of the existing components and technology stack.

We experimented with different approaches for a rendering XML in the web application, including:

1. Placing XML directly inside an HTML document and render it though the same CSS that we use now
2. Using the Canvas to display the rendered XML document, similar to how it is done inside oXygen, using a CSS parser that will provide the rendering styles for each element
3. Render/convert the XML as/to HTML and convert the CSS used for XML to match the converted HTML format

In the first case we hit limitations in the browser support for CSS that made it impossible to use this approach. For example browsers do not support the CSS `attr/2` function as specified in CSS3, where along with the first parameter that specifies the attribute name you can specify also a second parameter that represents the attribute value type. This is used in oXygen to specify that an attribute value is a URI and it should be a link.

In the second case we implemented all the rendering primitives that are used in oXygen (we have a Graphics interface that is used for rendering the XML documents and the methods from this interface were implemented also based on the HTML5 Canvas) but then we needed also the CSS parser, the layout engine, caret management, etc. which were not easy tasks.

In the 3rd approach we converted the XML document to HTML5 and then we modified the CSS that matched on XML to match on the converted HTML5 structure to obtain the same rendering as the XML+CSS that we currently use. This allows us to use the browser editing support for HTML to modify the document content.

For mobile interaction we use JQuery mobile due to existing experience - we use this also for the mobile-friendly WebHelp transformations that we provide for DITA and DocBook. However, other frameworks may be used as we plan to support multiple templates for the user interface.

3. Web application architecture

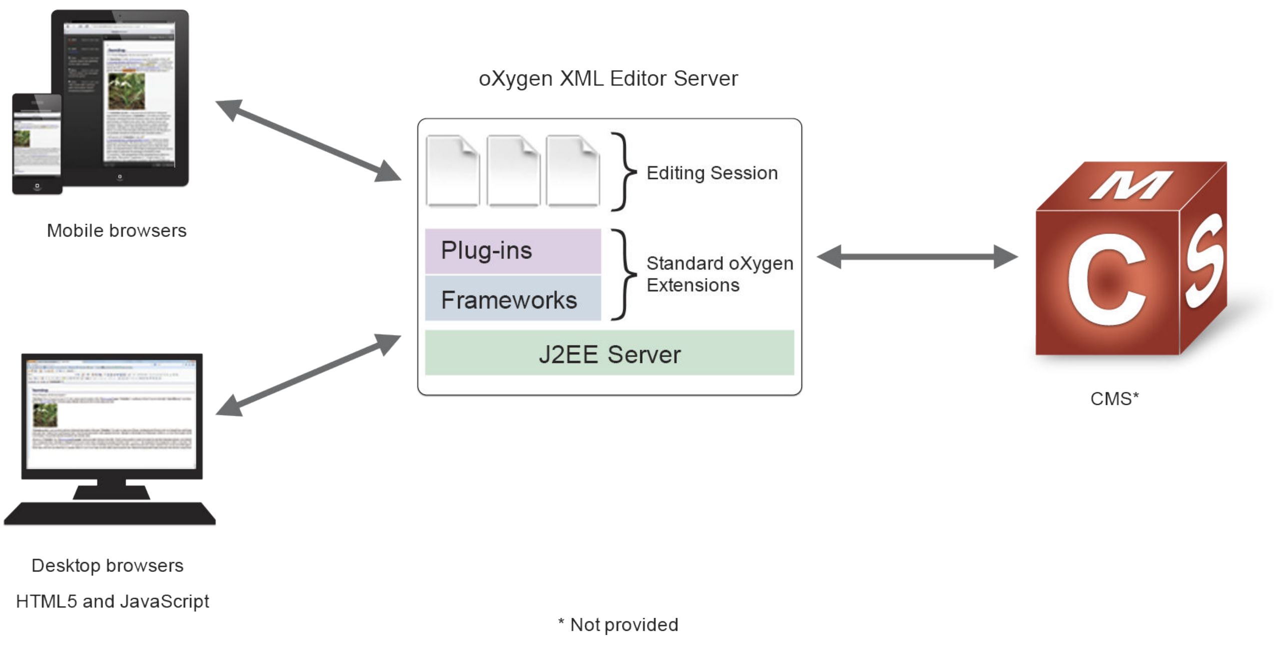
In Figure 4. oXygen web application architecture a diagram showing the current architecture outling how the oXygen existing support is reused on the server side. Three components can be identified:

- oXygen on the server
- The HTML+JavaScript that render the document on the client side
- Content storage that can be in the form of a CMS

oXygen on the server is a Java servlet that encapsulates the Java-based oXygen to provide the visual editing support. It reuses the same customizations created for the oXygen desktop that come in the form or frameworks and plugins. This allows for example to reuse the editing support for DITA, DocBook, etc. as well as plugins that provide access to remote repositories like the CMS connector plugins.

The server part will generate HTML5+JavaScript for an XML file that when rendered will provide the view for that XML document. The generated HTML5 content keeps XML related information in `data-*` attributes. The CSS that matched on XML is automatically converted to match on the generated HTML5 and its `data-*` attributes that encode the XML information. The conversion from XML to HTML5 uses mainly `div` elements but sometimes it also takes advantage of specific HTML elements, like the table element for example.

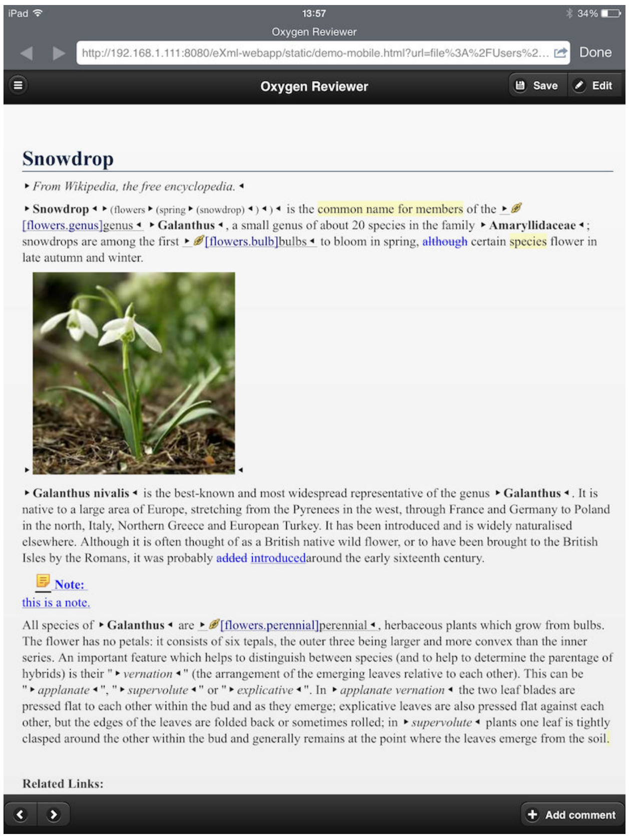
Figure 4. oXygen web application architecture



4. Samples

We will demo XML reviewing functionality, using custom XML interfaces and full XML editing. Here you can see some screen-shots taken on iPad:

Figure 5. A DITA topic


iPad
13:57
34%
Oxygen Reviewer
http://192.168.1.111:8080/eXml-webapp/static/demo-mobile.html?url=file%3A%2FUsers%2...
Done
Oxygen Reviewer
Save
Edit
Add comment

Snowdrop

From Wikipedia, the free encyclopedia.

Snowdrop (flowers (spring (snowdrop))) is the common name for members of the [flowers.genus]genus **Galanthus**, a small genus of about 20 species in the family **Amaryllidaceae**; snowdrops are among the first [flowers.bulb]bulbs to bloom in spring, ~~although~~ certain species flower in late autumn and winter.

Galanthus nivalis is the best-known and most widespread representative of the genus **Galanthus**. It is native to a large area of Europe, stretching from the Pyrenees in the west, through France and Germany to Poland in the north, Italy, Northern Greece and European Turkey. It has been introduced and is widely naturalised elsewhere. Although it is often thought of as a British native wild flower, or to have been brought to the British Isles by the Romans, it was probably ~~added~~ introducedaround the early sixteenth century.

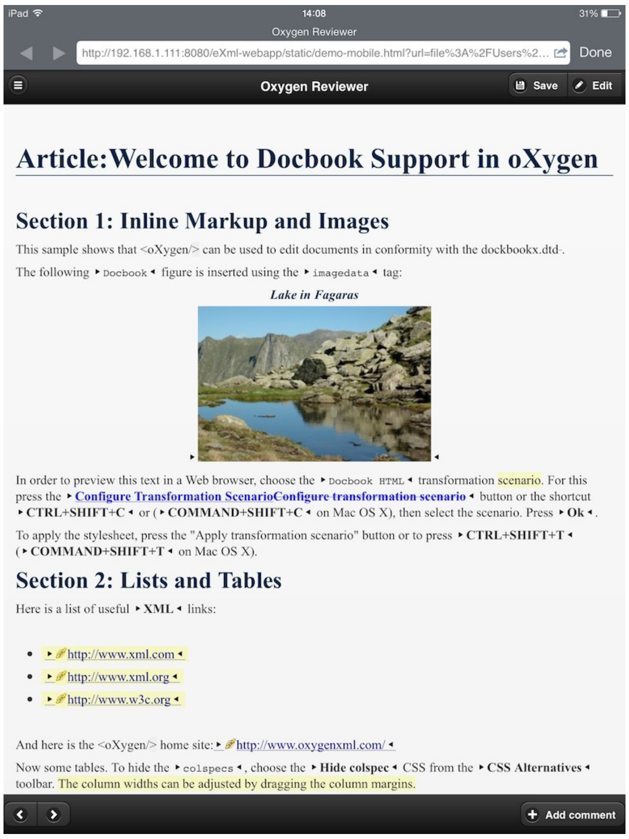
Note:
this is a note.

All species of **Galanthus** are [flowers.perennial]perennial, herbaceous plants which grow from bulbs. The flower has no petals: it consists of six tepals, the outer three being larger and more convex than the inner series. An important feature which helps to distinguish between species (and to help to determine the parentage of hybrids) is their "*vernation*" (the arrangement of the emerging leaves relative to each other). This can be "*applanate*", "*supervolute*" or "*explicative*". In *applanate vernation* the two leaf blades are pressed flat to each other within the bud and as they emerge; explicative leaves are also pressed flat against each other, but the edges of the leaves are folded back or sometimes rolled; in *supervolute* plants one leaf is tightly clasped around the other within the bud and generally remains at the point where the leaves emerge from the soil.

Related Links:

Note the highlights that represent areas with associated comments as well as the added and deleted content styled with underline and strikeout decorations.

Figure 6. A DocBook article


iPad
14:08
31%
Oxygen Reviewer
http://192.168.1.111:8080/eXml-webapp/static/demo-mobile.html?url=file%3A%2FUsers%2...
Done
Oxygen Reviewer
Save
Edit
Add comment

Article:Welcome to Docbook Support in oXygen

Section 1: Inline Markup and Images

This sample shows that <oXygen/> can be used to edit documents in conformity with the dockbookx.dtd-.

The following `Docbook` figure is inserted using the `imagedata` tag:

Lake in Fagaras

In order to preview this text in a Web browser, choose the `Docbook HTML` transformation scenario. For this press the **Configure Transformation Scenario~~Configure transformation scenario~~** button or the shortcut **CTRL+SHIFT+C** or (**COMMAND+SHIFT+C** on Mac OS X), then select the scenario. Press **Ok**.

To apply the stylesheet, press the "Apply transformation scenario" button or to press **CTRL+SHIFT+T** (**COMMAND+SHIFT+T** on Mac OS X).

Section 2: Lists and Tables

Here is a list of useful **XML** links:

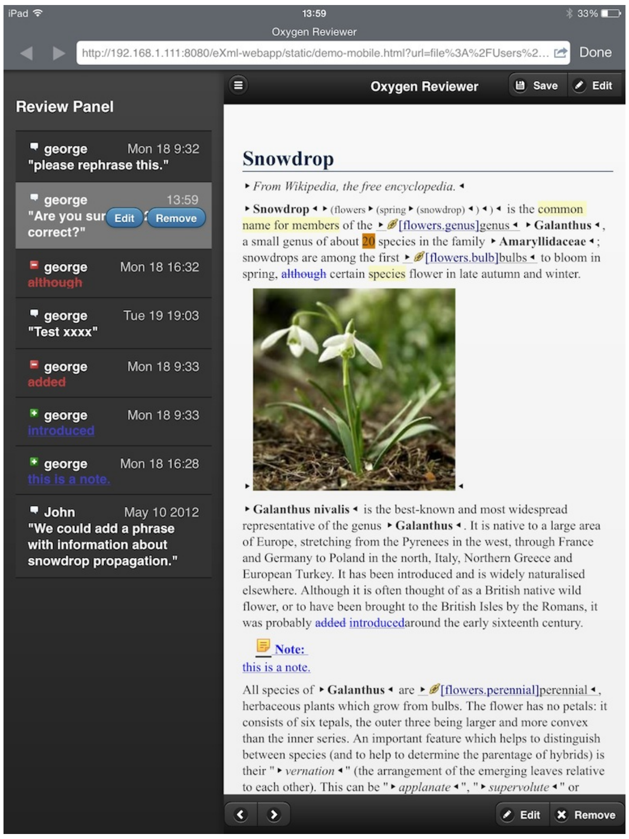
- http://www.xml.com
- http://www.xml.org
- http://www.w3c.org

And here is the <oXygen/> home site: http://www.oxygenxml.com/

Now some tables. To hide the `colspecs`, choose the **Hide colspec** CSS from the **CSS Alternatives** toolbar. The column widths can be adjusted by dragging the column margins.

Here we have a DocBook article rendered though CSS with different structure like images and lists. Note again the comment highlights and the decorations for changed content.

Figure 7. Review Panel showing all review comments and changes


iPad
13:59
33%
Oxygen Reviewer
http://192.168.1.111:8080/eXml-webapp/static/demo-mobile.html?url=file%3A%2FUsers%2...
Done
Oxygen Reviewer
Save
Edit
Edit
Remove
Edit
Remove

Review Panel

george Mon 18 9:32
"please rephrase this."

george 13:59
"Are you sur[e] correct?"

george Mon 18 16:32
although

george Tue 19 19:03
"Test xxxx"

george Mon 18 9:33
added

george Mon 18 9:33
introduced

george Mon 18 16:28
this is a note.

John May 10 2012
"We could add a phrase with information about snowdrop propagation."

Snowdrop

From Wikipedia, the free encyclopedia.

Snowdrop (flowers (spring (snowdrop))) is the common name for members of the [flowers.genus]genus **Galanthus**, a small genus of about 20 species in the family **Amaryllidaceae**; snowdrops are among the first [flowers.bulb]bulbs to bloom in spring, although certain species flower in late autumn and winter.

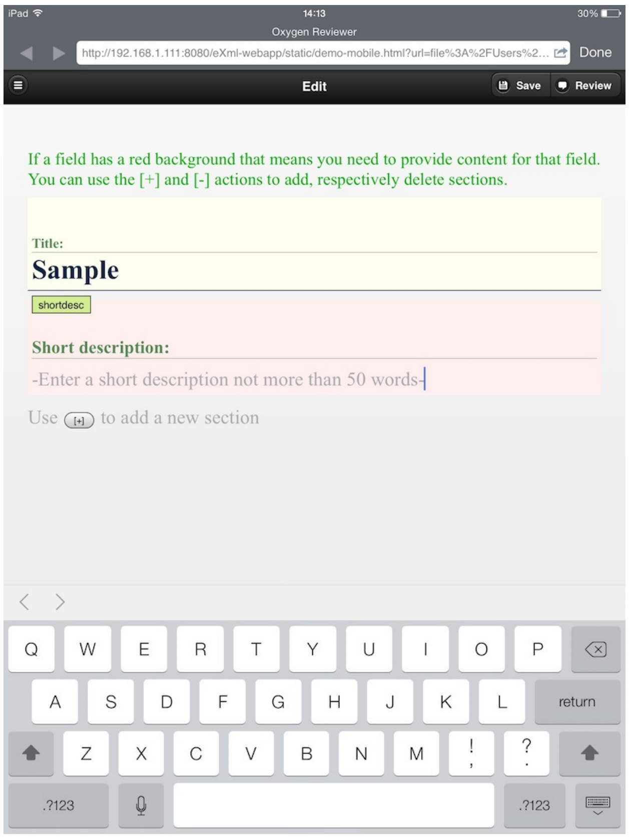
Galanthus nivalis is the best-known and most widespread representative of the genus **Galanthus**. It is native to a large area of Europe, stretching from the Pyrenees in the west, through France and Germany to Poland in the north, Italy, Northern Greece and European Turkey. It has been introduced and is widely naturalised elsewhere. Although it is often thought of as a British native wild flower, or to have been brought to the British Isles by the Romans, it was probably ~~added~~ introducedaround the early sixteenth century.

Note:
this is a note.

All species of **Galanthus** are [flowers.perennial]perennial, herbaceous plants which grow from bulbs. The flower has no petals: it consists of six tepals, the outer three being larger and more convex than the inner series. An important feature which helps to distinguish between species (and to help to determine the parentage of hybrids) is their "*vernation*" (the arrangement of the emerging leaves relative to each other). This can be "*applanate*", "*supervolute*" or

You can swipe right on the editing area to make the Review Panel visible. When you swipe over a review entry the available actions are displayed so you can easily act on a review to edit or remove a comment, accept or reject a change. Swipe left to hide the Review Panel and return to the editor.

Figure 8. A custom XML editing interface



If a field has a red background that means you need to provide content for that field. You can use the [+] and [-] actions to add, respectively delete sections.

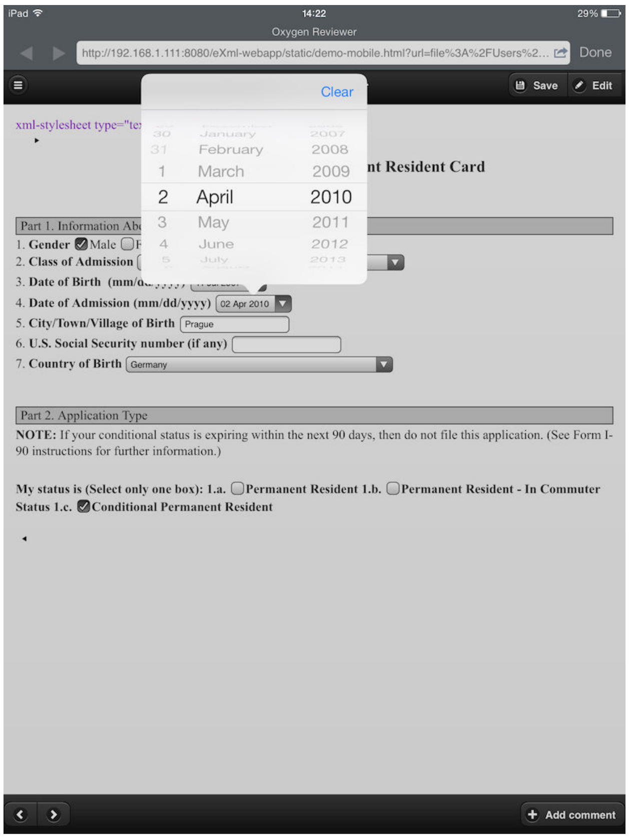
Title:
Sample

Short description:
-Enter a short description not more than 50 words-

Use [+] to add a new section

Note the inline action marked with "[+]" that can be used to add a new section to the document.

Figure 9. Enter a date value using the standard iPad date picker


iPad
14:22
29%
Oxygen Reviewer
http://192.168.1.111:8080/eXml-webapp/static/demo-mobile.html?url=file%3A%2FUsers%2...
Done
Save
Edit
Clear
Add comment

xml-stylesheet type="te

nt Resident Card

30	January	2007
31	February	2008
1	March	2009
2	April	2010
3	May	2011
4	June	2012
5	July	2013

Part 1. Information Ab

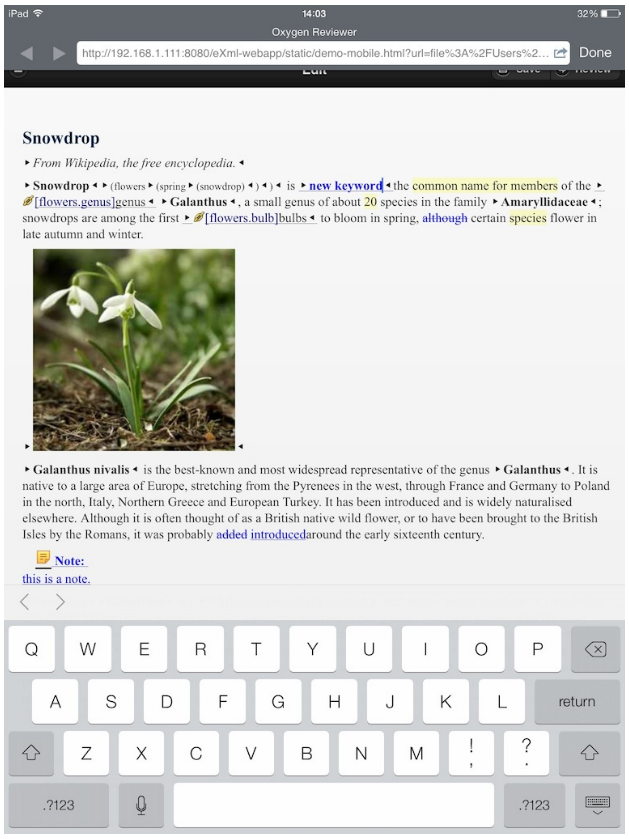
1. Gender [✓ Male] F
2. Class of Admission
3. Date of Birth (mm/d
4. Date of Admission (mm/dd/yyyy) 02 Apr 2010
5. City/Town/Village of Birth Prague
6. U.S. Social Security number (if any)
7. Country of Birth Germany

Part 2. Application Type

NOTE: If your conditional status is expiring within the next 90 days, then do not file this application. (See Form I-90 instructions for further information.)

My status is (Select only one box): 1.a. Permanent Resident 1.b. Permanent Resident - In Commuter Status 1.c. [✓ Conditional Permanent Resident]

Figure 10. Text editing with changes recorded as tracked-changes



Snowdrop

From Wikipedia, the free encyclopedia.

Snowdrop (flowers (spring (snowdrop))) is **new keyword** the common name for members of the [flowers.genus]genus **Galanthus**, a small genus of about 20 species in the family **Amaryllidaceae**; snowdrops are among the first [flowers.bulb]bulbs to bloom in spring, ~~although~~ certain species flower in late autumn and winter.

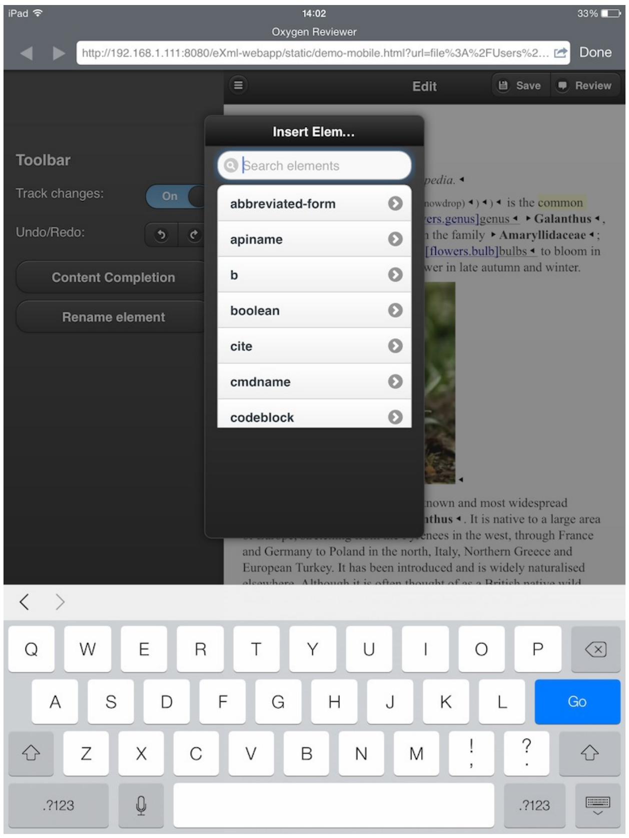
Galanthus nivalis is the best-known and most widespread representative of the genus **Galanthus**. It is native to a large area of Europe, stretching from the Pyrenees in the west, through France and Germany to Poland in the north, Italy, Northern Greece and European Turkey. It has been introduced and is widely naturalised elsewhere. Although it is often thought of as a British native wild flower, or to have been brought to the British Isles by the Romans, it was probably added introducedaround the early sixteenth century.

Note:
this is a note.

Different form controls can be used to build custom interfaces that will provide access to text and attribute values, thus making the editing simpler and removing the need to train users. Each form control will use the native support on each platform, thus the user will have the same editing experience he is already used to on that device.

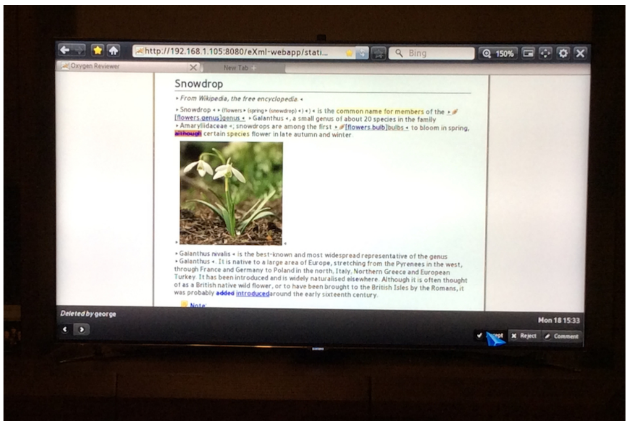
Here you can see the editing mode, where we have the keyboard show up and the document contains a caret. The changes in this case are recorded as tracked changes.

Figure 11. Inserting markup



You can insert markup either with the dedicated action or by pressing the enter key in the virtual keyboard. That will show a popup with valid element names where you can filter to see only the elements that match, then select one to insert in the document.

Figure 12. A DITA topic on a smart TV



The web editing platform works on any device supporting HTML5 and JavaScript, in this case we have it running on a smart TV.

5. Conclusions

XML editing on mobile devices can solve some real use-cases where people that are not XML-aware can contribute XML content using their preferred or available device at that moment. Creating a customized user interface using form controls bind to attribute values and inline actions reduce the training sometimes to zero - probably this is the way further, putting more effort on the developer to customize the user interface so that users will not have to think in terms of XML concepts but focus on the information they want to record.

There are large costs connected with integrating the reviewers feedback on some output format back into the XML source and sometimes that feedback is lost. A solution that will record reviewers feedback directly in the XML documents reduces dramatically the costs and effort and elliminates many steps in the workflow that can introduce errors.

There is a lot of exploration to come up with the best possible user interface that takes advantage of specific input methods and interaction patterns from mobile devices and this is just the start.

More generally, the web editing support for XML makes it available on any device, not only on mobile devices and it will be interesting to see if we can get different other applications based on XML like an XML-based blogging system or an XML-based wiki-like system.

Engineering a XML-based Content Hub for Enterprise Publishing

Elias Weingärtner

Haufe Group

Christoph Ludwig

Haufe Group

Abstract

Being one of the leading publishing houses in the domains of tax, human resources and law in Germany, delivering large amounts of XML-based content to our customers is a vital part of our business at Haufe Group. We currently make use of several legacy and proprietary systems for this purpose. However, recent business needs such as the requirement for flexible transformation or complex structural queries push these systems to both conceptual and technical limits. Along with new business requirements derived from our company's business strategy, we are currently designing a new service that centrally manages our entire document corpus in XML. We term this service "Content Hub". In this paper, we sketch the architecture of this system, discuss important software architectural challenges and illustrate how we are implementing this system using standard XML technology.

1. Introduction

Fifteen years ago, books, newspapers and magazines still were the most prominent media for the presentation of textual content. In the year of 2014, e-book readers, smartphones, tablet PCs and laptops are commonly observed as the major devices used for delivering information in digital form. This radical shift in how we access, read and retrieve content has turned the requirements and processes at media houses upside down.

The delivery of content for business customers in the domains of tax and law has always been one of the core businesses of Haufe Group. Since the 1960s, our company has been distributing loose leaf editions. There is to this day a surprising demand for paper-bound editions. Nevertheless, Haufe reacted to the rise of the WWW and entered the market with web-based content products for professional customers in the late 1990s.

For the purpose of publishing digital content the Haufe Group has developed a both comprehensive and sophisticated information ecosystem. We currently operate a set of custom-built systems that cover all steps ranging from content production based on SGML over content transformation to content presentation.

Even if our systems are stable, there is a core problem with our current information infrastructure: Core services such as search, document storage and retrieval as well as content-based authorization are saturated across the entire system landscape. For example, this results in the need of indexing content at different search engines, in a lot of duplicated content and partially also in limitations with regard to possible sales models. For this reasons, we are currently rethinking the way we store, manage and retrieve content in general. Speaking of the content body, we now provide around 50 millions of hypertext documents to our customers; and this amount is steadily increasing.

For these reasons, we are currently designing a new core service that centrally manages the entire document body using a NoSQL XML store. In this paper, we first sketch the requirements of this system (Section 2). In a second step, we briefly describe the future architecture of this system (Section 3) and related technical challenges (Section 5). Possible implementation approaches are sketched in Section 4 before we conclude the paper in Section 6.

2. Requirements

The Content Hub will be a core service in Haufe's future application landscape and as such has to meet many stakeholders' requirements, both functional and non-functional.

doi:10.14337/XMLLondon14.Weingaertner01

2.1. Functional Requirements

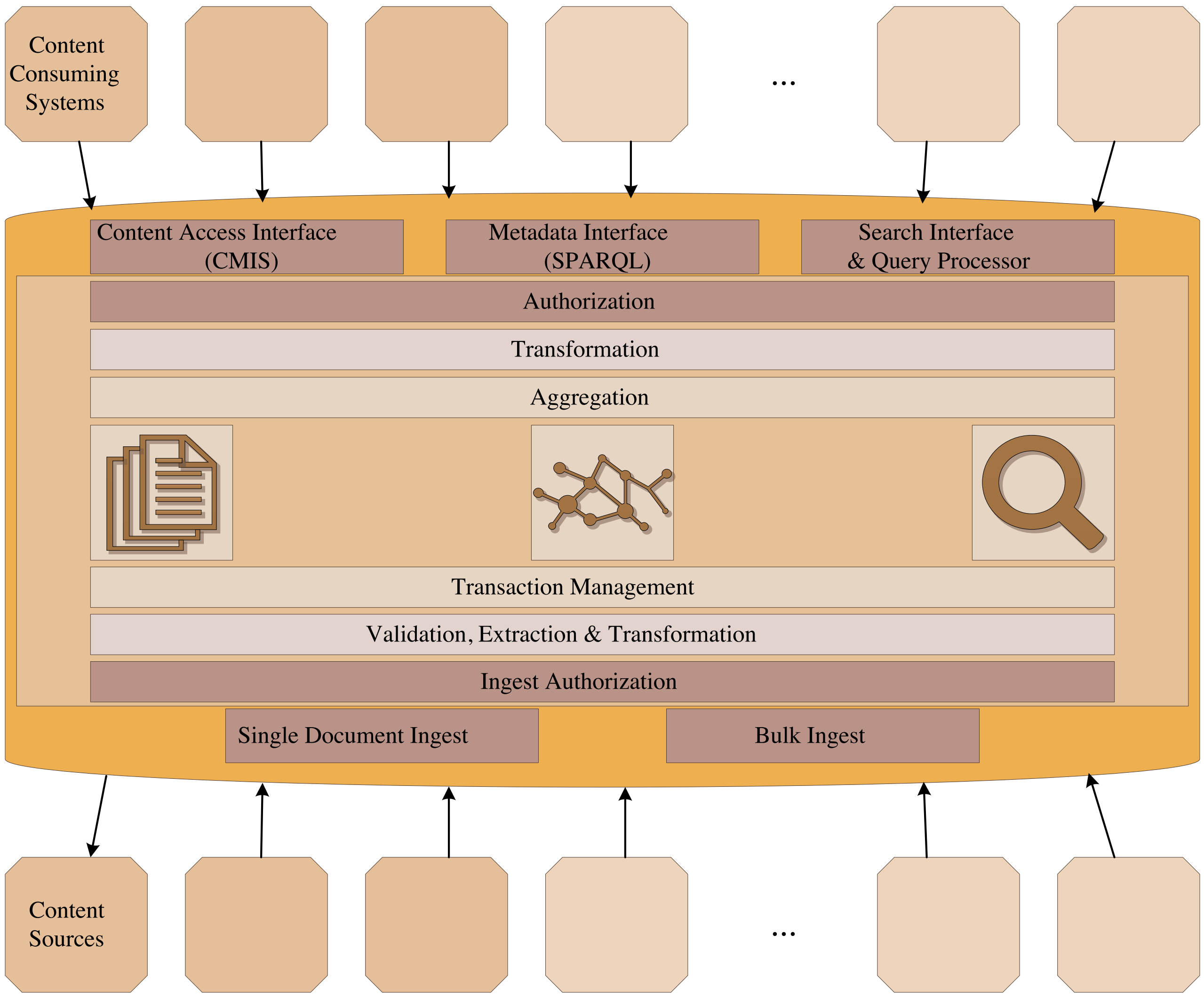
The following list can be understood as the set of minimal core functionalities that need to be implemented by our content hub.

- *XML and Blob Storage:* We envision the content hub to serve as the central store for all of our content. While we currently rely on SGML as source format for all of our 50 million documents, we are already have a working infrastructure for converting these documents to XML. Due to the higher flexibility of XML with regard to transformation and general tooling, we have decided to establish XML as the base line format for storing our documents in the content hub. Moreover, we are also in the need of storing a large amount of complimentary content (mostly images, but also software artifacts and audio-visual content). For this reason, it is vital for the content hub to be able to handle not only XML files, but also binary content.
- *Flexible Search Capabilities:* The most prominent way how our customers access content is by performing various search actions. As a matter of fact, the content hub has to support full-text searches across the entire document corpus. Naturally, this search service needs to support different languages and provide all features common to off-the-shelf search engines like support for facets and indexed meta-data. Beyond these common search functionalities, our products also offer domain-specific query constructs. For example, if a customer enters a query that matches a common citation rule, the corresponding document is returned directly if it is available in the datastore. Hence, the search component of the content hub must be able to check if a query matches a certain pattern, and if yes, a specialized query handler must be triggered in order to process the query adequately.
- *Semantic Relationships and Inference:* Our content is highly structured and we maintain many different types of relationships among our content objects. For example, imagine a law document displayed by a user. This law document may relate to other documents like comments or news articles and even to seminars offered by a affiliated partner company. Documents are also often not stored as a single file, instead they consist of sub documents, and we use relations to glue these documents logically together. Hence, it is vital that we can use XML technologies like XPointer, XLink and semantic annotations using RDF to model these semantic relationships in our content. Technology wise, it is vital that the underlying technology will allow us to traverse these semantic networks efficiently. We are also interested in performing inference tasks in a second step.
- *Staging Support:* Many of our internal processes require staging of large amounts of content. One example is publishing a new content product, which may consist of many thousand XML files. We here need staging support to facilitate internal review processes before the publication becomes publicly available. The actual publication must be an atomic operation. If the product's publication fails for some reason half the way through, then it must be rolled back completely.
- *Flexible Authorization Concept:* The Content Hub must restrict access to its documents based on policies. An obvious policy requires that a direct Haufe customer may only read content that is part of a product subscribed by the customer. This extends to search results as well; a user must only see search results he is allowed to retrieve. The policies must also cover end-users authenticated by key account customers or external partners. At the same time, Haufe is not always free to grant access as it sees fit, even for Haufe's own applications. Some content is used within license contracts that impose constraints on its usage. In consequence, the authorization policies must respect usage restrictions for individual documents.

2.2. Non-Functional Requirements

Being a core service of our information platform, the content hub needs to fulfill a set of non-functional requirements commonly found for network services [1]. For example, the system needs to scale horizontally as we expect larger amounts of content to be added dynamically to the system. We also require the system to achieve a high degree of availability. Finally, we require a strong degree of data consistency especially between the search indices and the data store, as we both want to avoid dead content or dead links in search result lists. Another important aspect is deployment flexibility. First, we require the future system to be easily deployable in a cloud environment. Second, we aim at a high degree of automation for common tasks such as horizontal scaling and content migration. Finally, we target the implementation of standardized operations interfaces in order to integrate the content hub with standard monitoring and logging tools.

Figure 1. Conceptual layered architecture of the future content hub.


Content Consuming Systems
...
Content Access Interface (CMIS)
Metadata Interface (SPARQL)
Search Interface & Query Processor
Authorization
Transformation
Aggregation
Transaction Management
Validation, Extraction & Transformation
Ingest Authorization
Single Document Ingest
Bulk Ingest
Content Sources
...

3. Conceptual Architecture

Figure 1 shows the future architecture of the content hub. The content hub provides three interfaces to external systems, for example web applications. These interfaces facilitate retrieving, searching and navigating through the document body.

- The *Content Access Interface* implements a standard CMIS interface, allowing convenient access for a variety of both legacy and off-the-shelf content management systems. We also consider amending the Content Access Interface with interfaces secondary to CMIS if our upcoming proof-of-concept prototype should indicate this need.
- A *SPARQL [2] -based Interface* will be used for all retrieval tasks related with meta data. Recall that our content is highly structured and that relations between content objects play a major role. Using RDF for modeling these semantic relations and employing SPARQL as query language provides us with the possibility to handle our complex document graph. Moreover, we expect SPARQL to be a door opener for future linked data applications.
- We are currently in the progress of defining a *search interface* . Its task is to enable querying the document body using full-text search and faceted search operations; the interface will also enable one to restrict the result set based on policies. At present, we also are discussing how we can include contextual information such as the active product, the location of the user and its language into the information passed to the system as query. The *search interface* will be enhanced by a custom *query processor* . It will support query configurative extension hooks based on the domain conventions of the respective products' target audiences.

Similarly, the content hub needs to implement two interface to ingest XML documents from a larger number of sources. Here we need to distinguish between the ingest of single documents and bulk imports that may range up to hundreds of thousands of documents at once.

3.1. Core Components

Internally, the content hub consists of six "sandwich" layers that span across the three core building blocks of the content hub, namely a XML document store, a triple store and a full-text search engine (denoted by the large icons in the diagram). Directly on top of these layers sits a thin *Aggregation Layer* that serves as a unified facade and combines the search, retrieval and triple-store related functionalities in one place. The upper two layers are of higher complexity:

- The *Transformation Layer* converts the XML-based content to a set of intermediate and target formats, for example XHTML, EPub or PDF. Depending on the target format, the transformations could be implemented by means of XSLT, XML-FO, or more specific rendering mechanisms. In this regard we also want to emphasize that the content-hub only performs transformations directly related to the content it stores. Further transformations required for the content delivery will be performed at the content presentation services, for instance by a web content management system connected to the content hub.
- As mentioned before, restricting the access based on authorization policies is vital for our business. Hence, we envision a central *Authorization Layer* that is able to control content search and retrieval for every request sent to the system. Within the current phase of system design, XACML 3.0 [3] seems to be a very promising candidate for this task. Regarding the need of *authentication*, we are currently evaluating the integration of different authentication providers, for example OAuth 2.0.

In an analog way, we perform both authorization operations and content transformations also for incoming content in the *Ingest Authorization* and in a *Ingest Transformation Layer* . This layer also performs validations based on XML schema to reject malformed content. In addition, we here also extract meta-data and potentially carry out automated RDF annotations.

Finally, a *Transaction Management* layer ensures the data integrity among the core building blocks. This is especially important for bulk updates, for which we are in the need of performing roll-backs if adding or updating operations fail in between a bulk operation.

4. Implementation Considerations

Having sketched the overall architecture of the system, we are left with three strategies how such a system can be implemented.

1. *Build everything from scratch:* In the past, a lot of infrastructure-heavy systems, for instance a legacy document store, have been built in-house, mostly using Java and Python/Zope technology.
2. *Integrate an XML-Database, a triple-store and an open-source search engine such as Elasticsearch:* While this solution is certainly viable, a core challenge is maintaining data consistency among these three systems. Especially implementing transactional semantics on a distributed system is challenging, as this requires distributed snapshots to be taken for the purpose of enabling conditional roll-backs.
3. *Use a enterprise NoSQL datastore with XML-capabilities:* In order to circumvent consistency issues, an alternative approach is to rely on a NoSQL datastore that is able to handle large amounts of XML documents.

After sketching prototypes for each of the three options, we found option 3 to be the most appealing strategy. First, we can save tremendous development efforts, as we do not have to implement a lot of data-management functionalities. Second, we believe that off-the-shelf-solutions like MarkLogic Server [4] - which is our present No.1 candidate for the implementation - are much more stable than any home-brew implementation will ever be able to. We here favour such enterprise solutions over open source XML stores due to the better availability of specialized consulting services. Thirdly, certain business requirements demand the content hub to execute in a cloud environment. Modern NoSQL stores such as MarkLogic already are well prepared for such deployments.

5. Further Technical Challenges

There are different technical challenges that are very decisive success criteria for such a system. First, we need to integrate *external systems* such as authentication providers and in-house systems for license management. We are currently investigating if we can achieve this using XQuery and REST calls or if we need to rely on the Java API of MarkLogic to carry out these tasks.

Second, there is *performance* . As indicated earlier, we envision the content hub to serve several hundreds to a couple of thousands requests a second. Hence, we need to consider different strategies for operating the content hub in a cluster in order to ensure availability of the service even under high-load or in the case of node failure. In addition, we currently develop caching strategies to be implemented at different layers in the system.

The third crucial aspect is the *data model* used to store our content, as the data model directly influences the performance of any database technology. We are currently in a requirements engineering phase in order to further clarify business demands; we will use the outcome to revise the first draft of our data model that we have developed over the past months.

6. Summary and Outlook

We have sketched a preliminary architecture for a content hub that is presently designed to serve as the central repository for XML and Blob content at Haufe Group. We here envision a combination of a NoSQL XML store, on-the-fly content transformation, enterprise search capabilities and a triple store to form a more flexible backbone for digital publishing that also opens up new business opportunities. Within the next months, we will develop a proof-of-concept implementation to further investigate the concept. We look forward to discussing our present and upcoming ideas and questions with the audience at the conference.

References

[1] *Distributed Systems: Principles and Paradigms*. 1st Edition. Prentice Hall, 2001.

[2] *SPARQL 1.1 Overview*. March 2013.
Online Resource: http://www.w3.org/TR/sparql11-overview/ (accessed 03/2014)).

[3] *eXtensible Access Control Markup Language (XACML) Version 3.0* . January 2013.
Online Resource: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (accessed 03/2014)).

[4] *Inside MarkLogic Server*. MarkLogic Corporation. 2013.

A Visual Comparison Approach to Automated Regression Testing

Celina Huang

Antenna House, Inc.

<celina@antennahouse.com>

Abstract

Antenna House Regression Testing System (AHRTS) is an automated solution designed to perform visual regression testing of PDF output (PDF to PDF compare) from the Antenna House Formatter software by converting a set of baseline PDFs and a set of new PDFs to bitmaps, and then comparing the bitmaps pixel by pixel. Several functions of the system make use of XML and the final reports are generated using XML and XSL-FO. This paper addresses the importance of PDF to PDF comparison for regression testing and explains the visual comparison approach taken. We explain the issues of traditional methods such as manual regression testing and why the need for an automated solution. We also look at how AHRTS works and discuss the benefits we've seen since using it internally to test new releases of our own software. Given its visual-oriented capabilities, we then explore other possible uses beyond the original design intent.

Keywords: regression testing, PDF, XML, XSL-FO

1. Introduction

We developed the Antenna House Regression Testing System to meet our own internal need to test releases of our formatting software to make sure that what worked before still worked in the new releases. In developing the system, we arrived at a set of design requirements. These included:

- The system had to visually compare PDFs. Comparing the internal code would not work because differences in the internal structure of a PDF could still produce the same visual output, e.g., tagged PDF and PDF 1.5.
- A reasonable throughput speed was important. Our test suite of 10,000+ pages was taking several people up to 3 days to manually regression test by looking at the PDF pages side by side. We wanted to speed up the process and reduce the number of people and effort required.
- Ease of usability, with a graphical user interface and application programming interfaces, so both the Developers and the Support Team that did regression testing could use the tool.
- Where possible, we wanted to use our own tools, XML and XSL-FO.
- Reports needed to be meaningful, easy to navigate, customizable (this is where the XML and XSL-FO fits), and that were not overly verbose.

Our need to regression test is the same as for any software developer. As software evolves and improves overtime, the possibility of newly, unwanted behavior occurring always exists. Every time software is upgraded or a system is changed in some way, regression tests need to be run to ensure that the changes do not introduce new faults [1] and that the intended results are still being produced. Having been named "an unsung hero of the software testing world" by Jean Hartmann (Microsoft's test architect) [2], regression testing is truly vital to guarantee an error-free product and should never be overlooked. Providing valued customers with significant enhancements is of no benefit if the upgrade process manages to break features that were working successfully in previous versions of the software.

There are many methods to regression testing that can be performed, but in this paper, we will focus on visual regression testing which involves comparing outputs visually to determine if the new document matches, or deviates, from the reference document. For organizations producing PDF documents from XML, this is a sure way to detect problems such as missing content or broken graphics that might result from a system upgrade or change. In fact, this is the way Antenna House performs regression testing on new releases of our formatting software.

doi:10.14337/XMLLondon14.Huang01

The traditional method to regression testing of a formatted paged output is to visually compare two documents side-by-side and page-by-page to ensure changes in the software has not disrupted the production process in any way. [3] Test cases can vary between a subset and a large collection of documents. The test cases that we collected over the years are almost all customer files that provide a good sampling of formatting jobs and truly exercise features of our formatting engine, AH Formatter. We now have an extensive collection of files that are rerun as regression tests and have become the means for verifying that our software is behaving correctly. It's important to note the issues with manual regression testing:

- It is a tedious task that testers have to endure and overtime, the level of attention to detail will decline causing unwanted changes to go unnoticed.
- It is extremely time consuming (especially if the outputs being tested are large files) and results in delayed product releases if the tests are not completed on time.
- It can be costly due to the amount of resources and time it takes to do the regression testing.
- It is unreliable and often leads to subtle errors that are left undetected by the naked eye.
- It may not be done frequently enough due to how much time, money, and human effort it takes to successfully complete, which results in testing only on candidate release versions of the software.

Because our software required testing on a large scale, where there is a massive volume of information and repetitiveness, we needed to migrate to an automated process. It would make the process faster, more efficient, reduce the risk for human error, and increase the overall quality of the test. The challenge was finding an automated tool that would compare outputs on a visual level, as opposed to the underlying code, and also be able to handle the PDF to PDF comparison on a large scale. Due to these unique requirements, we could not find a suitable tool and thus, Antenna House set out to custom develop an automated solution to be used internally to test the output from new releases of our software.

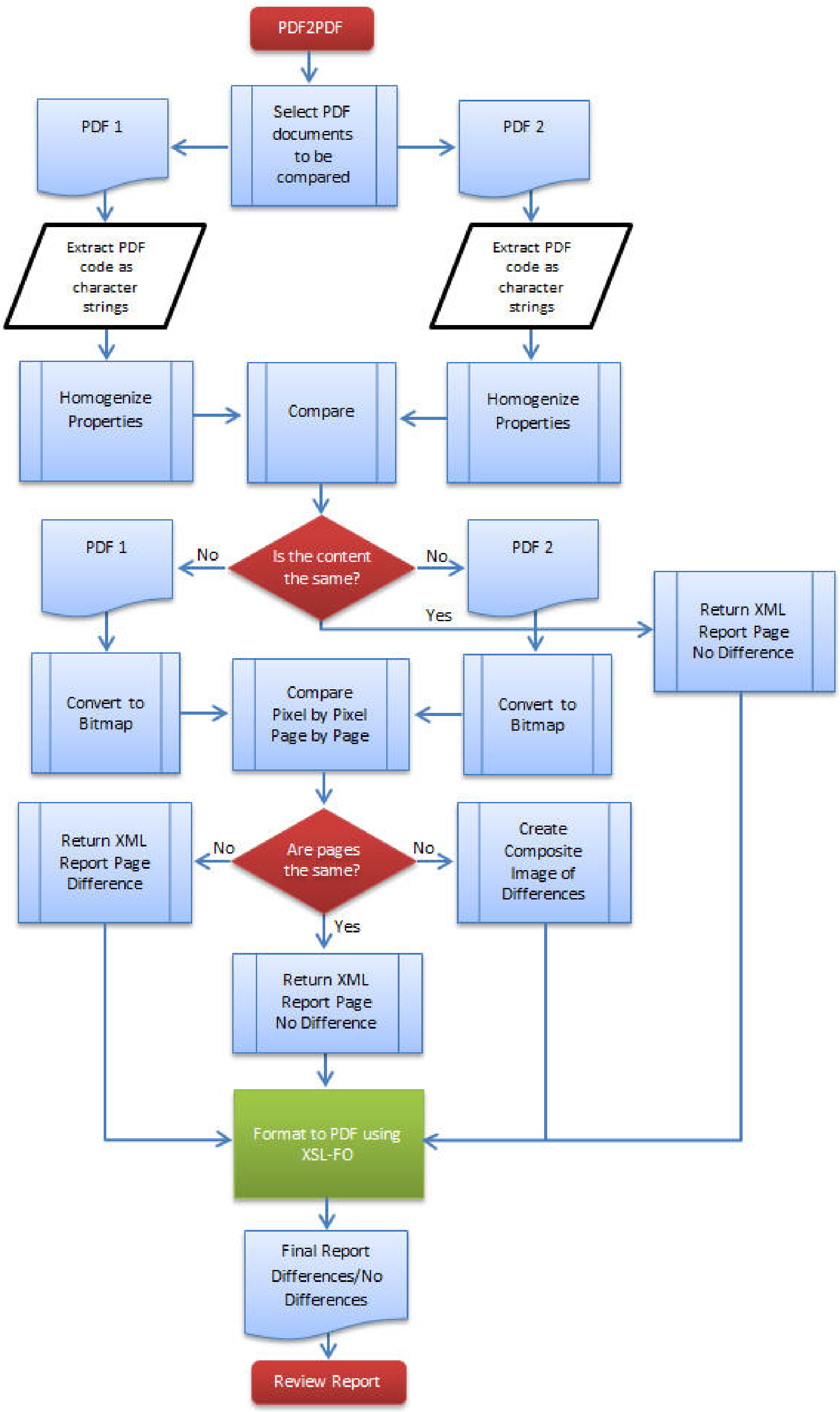
2. Automated Visual Regression Testing

The Antenna House Regression Testing System (AHRTS) is a simple, fast, and scalable solution for automating the visual comparison of formatted documents or pages. Its main function is to identify the visual dissimilarities that exist between a pair or set of documents. It is important to note that this is not a solution for tracking changes or locating variations in the content of multiple versions of the same document. However, it is capable of finding content discrepancies within a limited set of parameters. AHRTS is unique in that it quickly converts the PDFs to bitmaps and then performs a precision pixel-by-pixel comparison of two PDF files and generates a user friendly report that quickly identifies changes between a set of PDF files.

2.1. A Visual Method

Converting the PDFs to pixels and comparing pixel-to-pixel overcomes the issues of just looking at the underlying page codes. It also mitigates issues involved in testing multi-lingual documents and documents with vectors and bitmaps. For most organizations producing PDFs from XML, the most important thing is that the final output looks the same as it did before the software was altered. Minute differences as the name implies, might be small, but they can have a serious impact on the accuracy of a document. On graphics, lines, labels, fills, end caps might change or disappear. Text within the document might move unexpectedly. We believe that only a pixel-to-pixel comparison can locate even the most subtle of differences between two PDF documents. A detailed flowchart of how AHRTS works can be seen in Figure 1, “PDF2PDF Compare Flowchart”.

Figure 1. PDF2PDF Compare Flowchart


PDF2PDF
Select PDF documents to be compared
PDF 1
PDF 2
Extract PDF code as character strings
Extract PDF code as character strings
Homogenize Properties
Compare
Homogenize Properties
PDF 1
No
Is the content the same?
No
PDF 2
Yes
Return XML Report Page No Difference
Convert to Bitmap
Compare Pixel by Pixel Page by Page
Convert to Bitmap
Return XML Report Page Difference
No
Are pages the same?
No
Create Composite Image of Differences
Yes
Return XML Report Page No Difference
Format to PDF using XSL-FO
Final Report Differences/No Differences
Review Report

The process of comparing two PDF files starts with extracting the PDF code as character strings and then doing a checksum comparison to see which files need to be further tested. If document pairs with the same checksums are found, they are the same and will not be tested any further. If the documents differ, then the system will render them to bitmaps and then checksum compare each set of pages. Only the pages with different checksum are then compared at the pixel-to-pixel level. The differences found will be used to create a composite image and included in the final report along with the corresppondìng pages of the original PDF files being compared. Originally, our 10,000+ page test suite took close to 3 days of machine time to run. Once we implemented the two different checksum comparisons, the time to run the same test suite was reduced to less than 2 hours.

2.2. Handling Large Document Comparisons

Individual documents can vary from one page up to thousands of pages and collections of documents can be extremely large. AHRTS is able to handle document testing of any size, scaling from individual PDFs to directories of PDFs. Comparing multiple PDFs is no different from comparing individual PDFs when using AHRTS. Testers can choose the folders that need to be tested and a batch process will be performed to compare all of those documents in the specified folders. This system has no limit on the number of PDF files, the size of each file, or the number of pages being tested. We have actually tested a batch run of over 1,000 documents, containing a total of over 10,000 pages, without encountering any problems. A single PDF document with over 100,000 pages has also been tested with no issues at all. Being able to handle PDF documents of any size not only makes this system versatile to fit into different workflows, but also greatly reduces time and human resources devoted to regression testing.

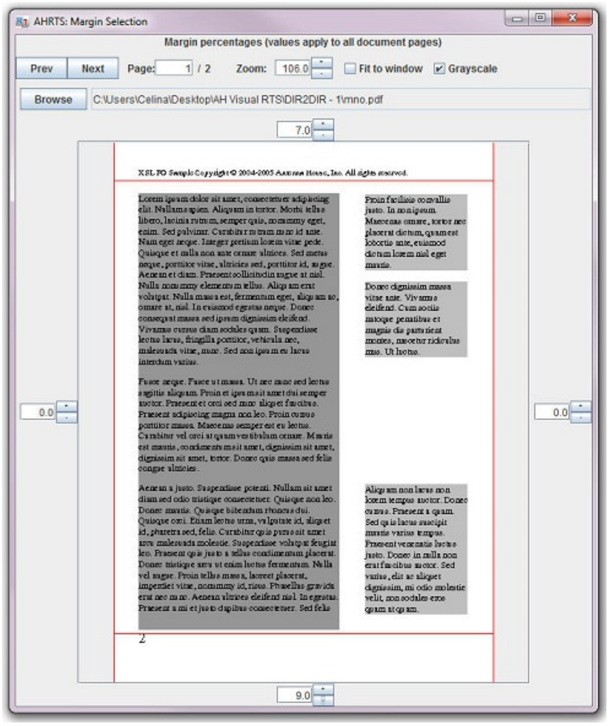
2.3. High Speed Performance

A valuable system is one that is able to regression test large collections quickly. In order to achieve such speed, AHRTS does a checksum comparison first (as seen in Figure 1, “PDF2PDF Compare Flowchart”), to detect which files are identical and which are not. When doing the checksum comparison, we do exclude some information such as the creation date and time of the PDF, which will have no bearing on the actual page output. Then only the documents with different checksums will move to the next testing phase where the discrepancies are identified. When AHRTS is used to test different versions of AH Formatter, two stages in the process can be set to multi-thread to attain even greater speed.

2.4. Exclude Margins from Testing

This is a new feature that allows us to exclude portions of the page in the trim and bleed areas that do not need to be tested. We can select an area from the top, bottom, left and/or right margin of the page to exclude from testing. A preview of individual pages being tested will appear in a separate window, as shown in Figure 2, “Margin Selection Preview Window”, to see what will be excluded as the margins are adjusted. You might want to use this for example, if the only known differences between the documents are an automatically generated date-time stamp.

Figure 2. Margin Selection Preview Window



Margin percentages (values apply to all document pages)

Page: 1 / 2 Zoom: 106.0 [ ] Fit to window [✓] Grayscale

C:\Users\Celina\Desktop\AH Visual RTS\DIR2DIR - 1\mno.pdf

7.0

0.0

0.0

9.0

XSL FO Sample Copyright © 2004-2005 Antenna House, Inc. All rights reserved.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nullam sapien. Aliquam in tortor. Morbi tellus libero, lacinia rutrum, semper quis, nonummy eget, enim. Sed pulvinar. Curabitur rutrum nunc id ante. Nam eget neque. Integer pretium lorem vitae pede. Quisque et nulla non ante ornare ultrices. Sed metus neque, porttitor vitae, ultricies sed, porttitor id, augue. Aenean et diam. Praesent sollicitudin augue at nisl. Nulla nonummy elementum tellus. Aliquam erat volutpat. Nulla massa est, fermentum eget, aliquam ac, ornare at, nisl. In euismod egestas neque. Donec consequat massa sed ipsum dignissim eleifend. Vivamus cursus diam sodales quam. Suspendisse lectus lacus, fringilla porttitor, vehicula nec, malesuada vitae, nunc. Sed non ipsum eu lacus interdum varius.

Fusce neque. Fusce ut massa. Ut nec nunc sed lectus sagittis aliquam. Proin et ipsum sit amet dui semper auctor. Praesent et orci sed nunc aliquet faucibus. Praesent adipiscing magna non leo. Proin cursus porttitor massa. Maecenas semper est eu lectus. Curabitur vel orci at quam vestibulum ornare. Mauris est mauris, condimentum sit amet, dignissim sit amet, dignissim sit amet, tortor. Donec quis massa sed felis congue ultricies.

Aenean a justo. Suspendisse potenti. Nullam sit amet diam sed odio tristique consectetuer. Quisque non leo. Donec mauris. Quisque bibendum rhoncus dui. Quisque orci. Etiam lectus urna, vulputate id, aliquet id, pharetra sed, felis. Curabitur quis purus sit amet arcu malesuada molestie. Suspendisse volutpat feugiat leo. Praesent quis justo a tellus condimentum placerat. Donec tristique arcu ut enim luctus fermentum. Nulla vel augue. Proin tellus massa, laoreet placerat, imperdiet vitae, nonummy id, risus. Phasellus gravida erat nec nunc. Aenean ultrices eleifend nisl. In egestas. Praesent a mi et justo dapibus consectetuer. Sed felis

Proin facilisis convallis justo. In non ipsum. Maecenas ornare, tortor nec placerat dictum, quam est lobortis ante, euismod dictum lorem nisl eget mauris.

Donec dignissim massa vitae ante. Vivamus eleifend. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut luctus.

Aliquam non lacus non lorem tempus auctor. Donec cursus. Praesent a quam. Sed quis lacus suscipit mauris varius tempus. Praesent venenatis lectus justo. Donec in nulla non erat faucibus auctor. Sed varius, elit ac aliquet dignissim, mi odio molestie velit, non sodales eros quam at quam.

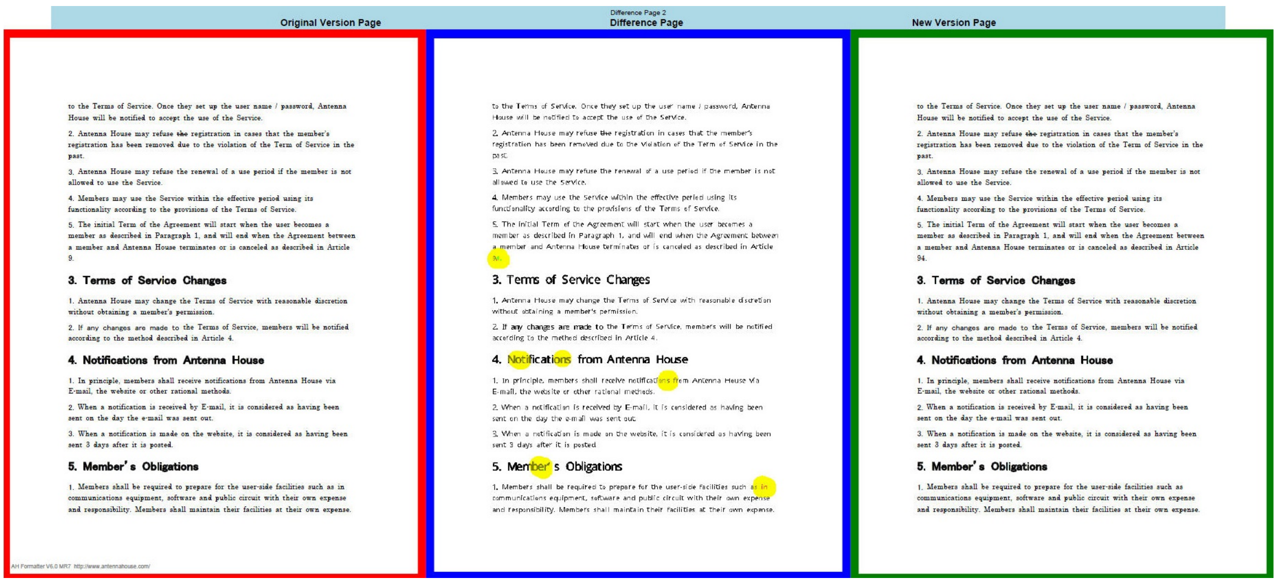
2

This option is also useful for certain documents that have updated content in the header or footer, but the rest of the content in the main body remains the same. By being able to exclude areas of the page, those intended differences would not cause every page of the document to be identified as different in the generated report.

2.5. Generates Usable Reports

After the comparison process is completed, an XML report is generated and formatted with AH XSL Formatter[1] to produce a usable PDF. The XSL stylesheet for the report produces a cover page that provides the total number of pages tested and locates which pages have differences. When testing multiple PDF files, an overview report is produced displaying which documents have differences and which documents are identical. Documents identified as having differences are hyperlinked to individual reports for each document. This individual report, as illustrated in Figure 3, "Individual Report", shows which pages are different and then presents a page composed of three panels for each pages with differences.

Figure 3. Individual Report



Original Version Page

to the Terms of Service. Once they set up the user name / password, Antenna House will be notified to accept the use of the Service.

2. Antenna House may refuse ~~the~~ registration in cases that the member's registration has been removed due to the violation of the Term of Service in the past.

3. Antenna House may refuse the renewal of a use period if the member is not allowed to use the Service.

4. Members may use the Service within the effective period using its functionality according to the provisions of the Terms of Service.

5. The initial Term of the Agreement will start when the user becomes a member as described in Paragraph 1, and will end when the Agreement between a member and Antenna House terminates or is canceled as described in Article 9.

3. Terms of Service Changes

1. Antenna House may change the Terms of Service with reasonable discretion without obtaining a member's permission.

2. If any changes are made to the Terms of Service, members will be notified according to the method described in Article 4.

4. Notifications from Antenna House

1. In principle, members shall receive notifications from Antenna House via E-mail, the website or other rational methods.

2. When a notification is received by E-mail, it is considered as having been sent on the day the e-mail was sent out.

3. When a notification is made on the website, it is considered as having been sent 3 days after it is posted.

5. Member's Obligations

1. Members shall be required to prepare for the user-side facilities such as in communications equipment, software and public circuit with their own expense and responsibility. Members shall maintain their facilities at their own expense.

AH Formatter V6.0 MR7 http://www.antennahouse.com/

Difference Page 2

Difference Page

to the Terms of Service. Once they set up the user name / password, Antenna House will be notified to accept the use of the Service.

2. Antenna House may refuse ~~the~~ registration in cases that the member's registration has been removed due to the violation of the Term of Service in the past.

3. Antenna House may refuse the renewal of a use period if the member is not allowed to use the Service.

4. Members may use the Service within the effective period using its functionality according to the provisions of the Terms of Service.

5. The initial Term of the Agreement will start when the user becomes a member as described in Paragraph 1, and will end when the Agreement between a member and Antenna House terminates or is canceled as described in Article 9.

3. Terms of Service Changes

1. Antenna House may change the Terms of Service with reasonable discretion without obtaining a member's permission.

2. If any changes are made to the Terms of Service, members will be notified according to the method described in Article 4.

4. Notifications from Antenna House

1. In principle, members shall receive notifications from Antenna House via E-mail, the website or other rational methods.

2. When a notification is received by E-mail, it is considered as having been sent on the day the e-mail was sent out.

3. When a notification is made on the website, it is considered as having been sent 3 days after it is posted.

5. Member's Obligations

1. Members shall be required to prepare for the user-side facilities such as in communications equipment, software and public circuit with their own expense and responsibility. Members shall maintain their facilities at their own expense.

New Version Page

to the Terms of Service. Once they set up the user name / password, Antenna House will be notified to accept the use of the Service.

2. Antenna House may refuse ~~the~~ registration in cases that the member's registration has been removed due to the violation of the Term of Service in the past.

3. Antenna House may refuse the renewal of a use period if the member is not allowed to use the Service.

4. Members may use the Service within the effective period using its functionality according to the provisions of the Terms of Service.

5. The initial Term of the Agreement will start when the user becomes a member as described in Paragraph 1, and will end when the Agreement between a member and Antenna House terminates or is canceled as described in Article 94.

3. Terms of Service Changes

1. Antenna House may change the Terms of Service with reasonable discretion without obtaining a member's permission.

2. If any changes are made to the Terms of Service, members will be notified according to the method described in Article 4.

4. Notifications from Antenna House

1. In principle, members shall receive notifications from Antenna House via E-mail, the website or other rational methods.

2. When a notification is received by E-mail, it is considered as having been sent on the day the e-mail was sent out.

3. When a notification is made on the website, it is considered as having been sent 3 days after it is posted.

5. Member's Obligations

1. Members shall be required to prepare for the user-side facilities such as communications equipment, software and public circuit with their own expense and responsibility. Members shall maintain their facilities at their own expense.

[1] A restricted copy of AH XSL Formatter is provided with AHRTS for only the purpose of generating reports.

The left pane is the actual original PDF page extracted from the baseline document and the right pane is the PDF page extracted from the new document. This is possible with AH XSL Formatter's ability to merge selected individual pages from a PDF into a single PDF file it creates. The center pane is the bitmap used for the pixel-to-pixel comparison with the added composite of the two pages highlighting where the changes are. There is an option to overlay the bitmap composite on top of the compared PDF files in the report for a better visual comparison. Color coding is used to further help identify what sort of change occurred between the original and new documents. If a portion of the page was intentionally excluded from testing, it can also be colored. It is important to note that only pages with differences are presented in each report. This way, if only 4 out of 500 pages have differences, you will only have to look at and scroll through those 4 specific page sets.

The XSL stylesheet that produces the reports can be modified to tailor the reports as wanted and all the XSL-FO capabilities of Antenna House Formatter are available for the report generation. This has enabled us to quickly try a large number of report layouts in order to arrive at the final report that best displays the information we want and the way we want it.

3. Regression Testing New Releases of AH Formatter

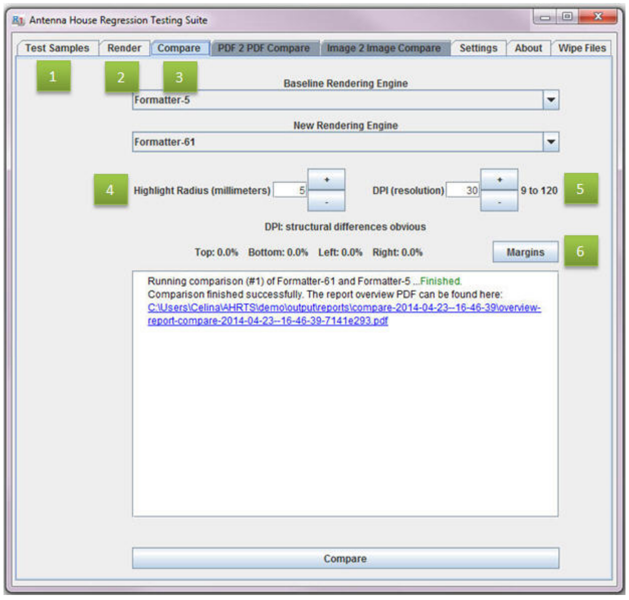
As previously mentioned, this software was originally developed for our own internal use to test new releases of AH Formatter. During the course of a year we will do close to 20 releases, which includes maintenance releases and new version releases of the versions of AH Formatter that are still being fully supported. What used to take several people several days, and was limited to only candidate releases of the software, now just takes a couple hours of machine time and is performed regularly on development versions as well. AHRTS has enabled us to detect possible negative regressions much earlier in the development cycle. It also resulted in more regular releases of AH Formatter, less associated problems in the new releases, and an overall better product.

So how does it work? When used to regression test releases of AH Formatter, AHRTS takes the documents and first creates a batch file. The batch file is then used to create the PDFs for both versions of AH Formatter that are being compared. Next, it does the comparison of the files and then creates all the associated reports. The final step is where a user has to review the finished reports.

3.1. The User Interface

This application can be launched via command-line interface or graphical user interface (GUI), but we will be focusing on the latter for the purposes of this paper and presentation. The GUI is written in Java so it will look the same whether you are using it on Windows, Linux, or a Mac operating system. A screenshot of the GUI with the "Compare" tab open can be seen below in Figure 4, "AHRTS User Interface".

Figure 4. AHRTS User Interface



Baseline Rendering Engine: Formatter-5
New Rendering Engine: Formatter-61
Highlight Radius (millimeters) 5
DPI (resolution) 30 9 to 120
DPI: structural differences obvious
Top: 0.0% Bottom: 0.0% Left: 0.0% Right: 0.0%

```
Running comparison (#1) of Formatter-61 and Formatter-5 ...Finished.
Comparison finished successfully. The report overview PDF can be found here:
C:\Users\Celina\AHRTS\demo\output\reports\compare-2014-04-23--16-46-39\overview-
report-compare-2014-04-23--16-46-39-7141e293.pdf
```

When the AHRTS GUI is first opened, it has default settings of where to locate and store files needed to complete the regression testing process. If AHRTS is integrated in a bigger system, users can change where they want the saved files to be stored in the "Settings" tab and reuse them in other applications in their workflow. The data for the reports generated is XML making it easily accessible.

The following features, numbered in Figure 4, "AHRTS User Interface", are the main steps to perform regression testing of AH Formatter:

1. **Test Samples**- We first create test cases by obtaining the original source files (.fo files) that Antenna House has collected over the years to make up our test suite. When AHRTS makes the test cases, it will store the XSL-FO file and include a XML file in a directory specified in the "Settings" tab, which was discussed earlier. It will also prepare new file names for the test cases for the Render and Compare steps.

2. **Render**- Next, we select the versions of AH Formatter to be added to a queue to render the test cases from a drop down menu. AHRTS uses engine files to locate and identify the different versions of Formatter that will be used with the system. There are premade engine files specified in XML format that will work for default installations of AH Formatter. If a version of AH Formatter is located in a different folder other than the default, users can manually change the directory path in the engine file so AHRTS can find it. This step is just rendering the test cases, so it is possible to run test cases through 3 or more different versions of AH Formatter. Since AH Formatter is thread safe, there is also an option to set the rendering jobs to multi-thread and enhance speed and performance. The number of threads set would depend on the number of processors or cores on the machine.
3. **Compare**- Once AHRTS confirms that each of the rendering engines have finished their jobs and produced PDF files, we can continue with the Compare step. In this demonstration, we chose Version 5 as the baseline engine and Version 6.1 as the new engine to be tested. AHRTS will follow the same process as illustrated in Figure 1, "PDF2PDF Compare Flowchart" in this stage. After the comparisons are completed, a new window will automatically open with a detailed PDF report to review. In both the "Compare" and "PDF2PDF Compare" tab, users can adjust the highlight radius, DPI settings, and margins to be excluded from testing.
4. **Highlight radius**- The yellow halo that highlights where the changes are in the composite image of the report can be increased or decreased in size. The default is a radius of 5 millimeters, but it can be set to 0 for no highlight at all or enlarged depending on a user's preference. Prior to the addition of the highlight feature, subtle differences such as the end of a line in a graphic were sometimes hard to locate in the color coded composite bitmap.
5. **DPI setting**- The Dots Per Square Inch (DPI) of a comparison is the resolution at which a PDF document from a rendering engine is rasterized. This setting allows the user to control the level of comparison and how sharp the composite page appears in the final report. With that in mind, a higher DPI means more pixels to compare per page and will take more time and power to finish the comparison process. The default is set at 30 DPI (900 dots per square inch), which we have found to be more than enough resolution to find differences in font types, minor alterations in spacing and line width, as well as larger ones such as differences in the breaking of pages.
6. **Margins setting**- This calls up Figure 2, "Margin Selection Preview Window", as previously discussed.

4. Other Possible Use Cases

Beyond being a regression testing system for new releases of AH Formatter, there are many other possible use cases for AHRTS, which we now discuss:

- Regression testing any document formatting or conversion software. A PDF to PDF comparison can be performed to regression test PDFs from virtually any source, as well as a growing list of image formats. This could include other formatting engines besides those provided by Antenna House, ranging from word processing, OCR, conversion, to business software. The only requirements are two PDFs or images to be compared.
- Pre-production system check: For documents that have extended periods between production such as an annual directory, PDF differencing can ensure that during the preceding period, no changes were made to the tool chain that adversely affect the previously good output.
- Stylesheet development: By comparing PDFs, you can quickly see if changes to the stylesheet actually achieved the desired results. For instance, did changing the thickness of a rule or bolding a heading cause type to overflow a block? Did shifting a line a couple points cause content to be lost or shifted in the wrong direction? Any of these unforeseen consequences that occur while developing a stylesheet, can be identified with a PDF to PDF comparison tool.

- Installation validation: Installing a new system where the final output is a document or publication, typically involves a lot of components that include an authoring or data source, a database or content management system, software to manipulate and assemble the data, and finally the tool that formats the output in a document. Each of these components in turn can have multiple, moving parts. Typically, a solution is implemented first on a development system and then quite often replicated on training systems, staging systems, QA systems, backup systems, and the actual production system. All of these systems need to produce exactly the same output, using the same tool chain and from the same source. AHRTS provides a way to take output that was successfully produced on the development or first system and compare that with output generated by each of the subsequently implemented systems.
- System(s) certification: In a multisystem environment where software is installed on more than one system, either locally or remotely, comparing PDFs helps validate that each system produces exactly the same document output from a collection of documents used for testing.
- Ensuring that different versions of PDF are producing the same visual output. It is conceivable that a document rendered to PDF/A or tagged PDF displays differently. At times, it would be important to know what those differences are.

5. Conclusion

In the beginning of this paper, we emphasized the importance of regression testing and how it plays a major role in any testing process in the software world. More specifically we're focusing on groups producing PDF documents from XML, like Antenna House, who need to test the visual output of their systems. Traditional methods proved to be unreliable, time consuming, costly, and overall a tedious task to say the least. Now with automation and using a visual comparison approach, AHRTS greatly increases the efficiency of regression testing, reducing the time and effort required to check output discrepancies, and enabling users to find even the slightest change in the visual appearance of a document. To achieve the visual PDF to PDF comparison we convert the PDFs to bitmaps and then compare them pixel to pixel. This produces a highly reliable check that will catch even the slightest differences between PDFs. Originally designed as a regression testing tool for AH Formatter, AHRTS has evolved into a tool that can play many roles in a development cycle for any system generating visual outputs.

Bibliography

[1] *The Art of Software Testing*. Glenford Myers. Wiley. December 2011. ISBN: 978-1-118-03196-4.

[2] *30 Years of Regression Testing: Past, Present and Future*. Jean Hartmann. http://www.uploads.pnsqc.org/2012/papers/t-67_Hartmann_paper.pdf

[3] *Antenna House: Antenna House Regression Testing System*. http://www.antennahouse.com/antenna-house-regression-testing-system/

Live XML Data

Steven Pemberton

CWI, Amsterdam

Abstract

XML is often thought of in terms of documents, or data being transferred between machines, but there is an aspect of XML often overlooked, and that is as a source of live data, that can be displayed in different ways in real time, and used in interactive applications.

In this paper we talk about the use of live XML data, and give some examples of its use.

1. Introduction

In [1], Tim Bray, one of the developers of XML, said

> "You know, the people who invented XML were a bunch of publishing technology geeks, and we really thought we were doing the smart document format for the future. Little did we know that it was going to be used for syndicated news feeds and purchase orders."

In other words, they did they not anticipate XML's use outside of documents and publishing, as data, as interactive documents, and so on.

But with the increasing availability of apps, live data is becoming more and more significant.

2. Live Data

Live XML data is the use of XML in an application where the data is constantly updated, either by repeated polling of an external source, or through interaction with the user, or a combination of both.

To give an example [2], it is currently good practice to give suggestions if a user is searching in a large database or similar. Using XML and XForms [3], [4], [5], [6], [7], it is easy to specify this: the search string is kept in instance data:

```
<root xmlns="">
  <search/>
</root>
```

which is input with an *incremental* control, that updates the data each time a key is pressed:

```
<input ref="search" incremental="true">
  <label>Search: </label>
</input>
```

Whenever the value is changed in the control (which invokes an `xforms-value-changed` event), the data is submitted to the site (in this case wikipedia):

```
<send ev:event="xforms-value-changed"
      submission="s1"/>
```

The submission that causes this specifies that the results of the submission are returned into a different instance

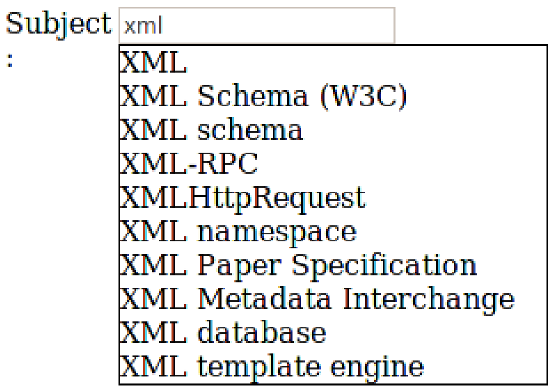
```
<submission id="s1" resource="
http://en.wikipedia.org/w/api.php?action=opensearch
" method="get"
  replace="instance"
  instance="iresults"/>
```

The results are then displayed to the user, as a series of 'triggers', which when clicked on, set the value of the search string:

```
<repeat id="results"
        nodeset="instance('iresults')/*[2]/*">
  <trigger appearance="minimal">
  <label><output value="."/></label>
    <action ev:event="DOMActivate">
      <setvalue
        ref="instance('isearch')/search"
        value="instance('iresults')
               /*[2]/*[index('results')]" />
    </action>
  </trigger>
</repeat>
```

doi:10.14337/XMLLondon14.Pemberton01

And the result looks like this:


Subject : xml

XML
XML Schema (W3C)
XML schema
XML-RPC
XMLHttpRequest
XML namespace
XML Paper Specification
XML Metadata Interchange
XML database
XML template engine

This is a general idiom that can be used in many places: source data is changed in some way, possibly by interactions from the user, and this causes data to be updated from external sources.

3. XForms

XForms is a language originally developed for dealing with forms on the web. However, thanks to the generality of its design it was soon realised that, with a little more generality, it could be used for more general applications as well. So since XForms version 1.1, applications can be built with XForms. In fact, a form is really just the collection of data, some calculation, and some output, as well as submission of data. But this is a actually the description of an application as well. The only real noticeable difference is the manner in which the data is collected and presented.

XForms has been in use for more than a decade now, by a wide range of users including the BBC, the Dutch national weather service, NASA, and Xerox, just to name a few. Experience has shown that XForms greatly reduces the time needed to produce an application (by about an order of magnitude). This is largely due to the approach used by XForms, of declaratively specifying *what* is to be achieved, rather than *how* to achieve it.

4. An Example

In XForms you can put the URL of an image in your data:

```
<instance>
  <data xmlns="">
    <url>
      http://tile.openstreetmap.org/10/511/340.png
    </url>
  </data>
</instance>
```

and output it with

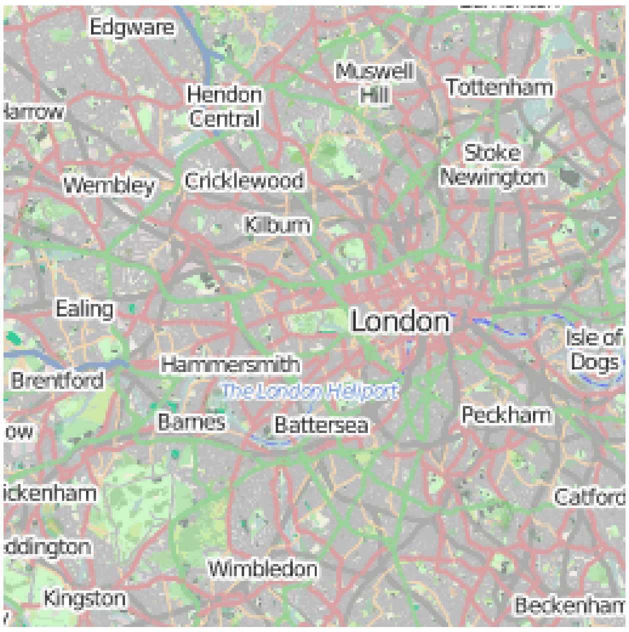
```
<output ref="url"/>
```

This would give as output:

```
http://tile.openstreetmap.org/10/511/340.png
```

But if you add a `mediatype` to the `<output>`, the image itself is output instead:

```
<output ref="url" mediatype="image/*" />
```



5. URL Structure

An Open Street Map URL is made up as:
`http://<site>/<zoom>/<x>/<y>.png`

So we can represent that in XForms data:

```
<instance>
  <map xmlns="">
    <site>http://tile.openstreetmap.org/</site>
    <zoom>10</zoom>
    <x>511</x>
    <y>340</y>
    <url/>
  </map>
</instance>
```

and calculate the URL from the parts:

```
<bind ref="url"
      calculate="concat(../site, ../zoom, '/',
                        ../x, '/', ../y, '.png')"/>
```

But now that we have the data, we can also input the different parts:

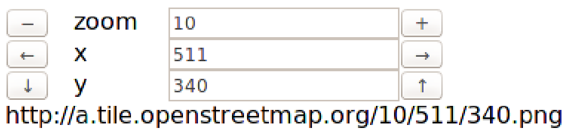
```
<input ref="zoom"><label>zoom</label></input>
```

This means that we can enter different values for the tile coordinates, and because XForms keep all relationships up-to-date, a new tile URL is calculated and the corresponding tile is displayed.

However, since entering numbers like this is inconvenient, we can also add some nudge buttons, of the form:

```
<trigger>
  <label>→</label>
  <setvalue ev:event="DOMActivate" ref="x"
            value=". + 1"/>
</trigger>
```

so it looks like this:



```
[–]  zoom  10   [+]
[←]  x     511  [→]
[↓]  y     340  [↑]
http://a.tile.openstreetmap.org/10/511/340.png
```

Edgware
Hendon Central
Muswell Hill
Tottenham
Harrow
Wembley
Cricklewood
Stoke Newington
Kilburn
Ealing
London
Isle of Dogs
Hammersmith
The London Heliport
Brentford
Barnes
Battersea
Peckham
Twickenham
Catford
Teddington
Wimbledon
Kingston
Beckenham

6. Zoom

A problem with this is that while the x and y nudge buttons work fine, the zoom button doesn't. This is because at each level of zoom the x and y coordinates change: at the outermost level of zoom, 0, there is one tile, x=0, y=0. At level 1, the coordinates double in both direction, [0-1], so there are 4 tiles; at level 2, the coordinates are [0-3], and there are 8 tiles, 16 at level 3, and in general 2z at level z (up to level 18).

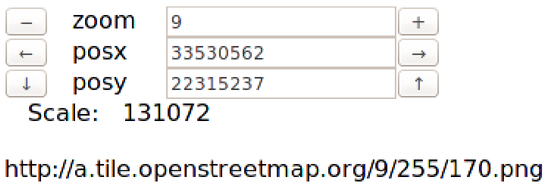
So to make zoom work properly, we must save our location in world coordinates, each value between 0 and 226 (which is the 18 levels of zoom, plus 8 bits for the 256 pixels of each tile), and then calculate the tile at any level of zoom from that:

```
scale=226 - zoom
    x=floor(posx/scale)
    y=floor(posy/scale)
```

In XForms:

```
<bind ref="scale"
      calculate="power(2, 26 - ../zoom)"/>
<bind ref="x"
      calculate="floor(../posx div ../scale)"/>
<bind ref="y"
      calculate="floor(../posy div ../scale)"/>
```

Now when you zoom in and out, the area remains the same:

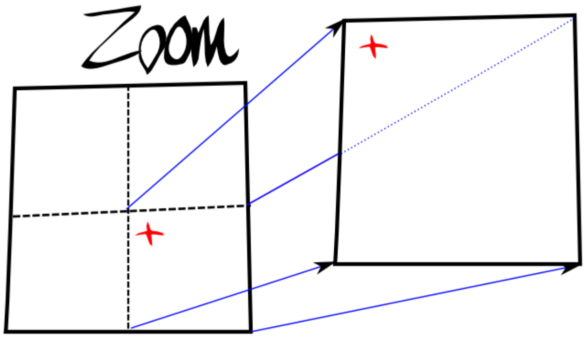


```
[–]  zoom  9         [+]
[←]  posx  33530562  [→]
[↓]  posy  22315237  [↑]
 Scale:  131072

http://a.tile.openstreetmap.org/9/255/170.png
```

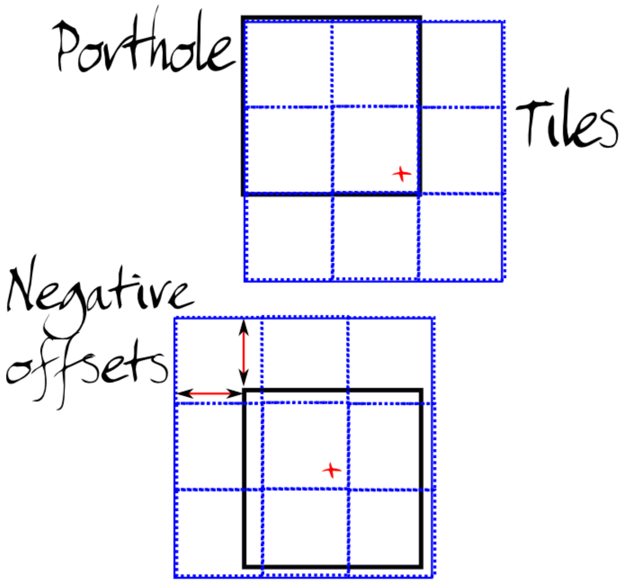
Wood Green
Denham
Harrow
Hendon
Uxbridge
Willesden
Hackney
Slough
Hayes
Ealing
London
Barnes
Peckham
Bedfont
Egham
Tooting
Ascot
Kingston
Beckenham
Weybridge
Croydon
Woking
Cobham
Epsom
Coulsdon
Leatherhead
Caterham
Guildford
Dorking
Redhill
Godalming

7. Location, location, location

You'll notice from the two images above that we got the tile that contains our location, but the location (in this case, central London) is at a different part of the tile. This is because if you have a tile where the location is in the middle of the tile, when you zoom in, you get one of the 4 quadrants, and so by definition, the location is no longer at the centre of the tile:

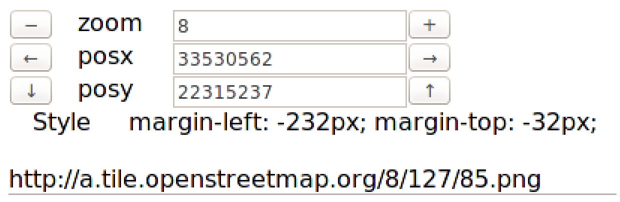


From a usability point of view of course, we want our location to remain in the middle of the view, so to achieve this, we create a 3×3 array of tiles, with a porthole over it. The porthole stays static, and we shift the tiles around underneath so that our location remains in the centre. This we do by calculating offsets that the tile array has to be shifted by, and then using these to construct a snippet of CSS to move the tile array:

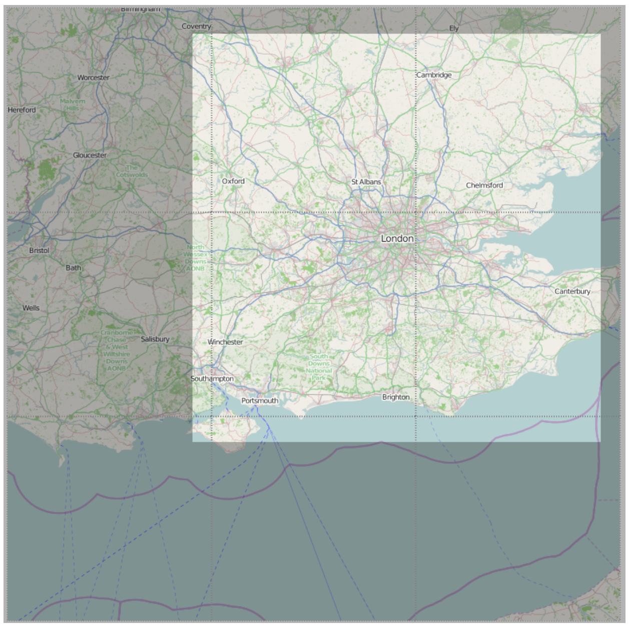


```
<bind ref="offx"
      calculate="0 - floor(((
                      ../posx - ../x * ../scale)
                      div ../scale)*../tilesize)"/>
<bind ref="offy"
   calculate="0 - floor(((
                    ../posy - ../y * ../scale)
                    div ../scale)*../tilesize)"/>
 ...
<div style="margin-left: {offx};
            margin-top: {offy}">
```

Now we have a live map, where we can zoom in and out, and pan left and right and up and down. Here is a view also showing the parts that would normally not be visible, outside of the porthole:



```
–  zoom  8         +
←  posx  33530562  →
↓  posy  22315237  ↑
Style  margin-left: -232px; margin-top: -32px;

http://a.tile.openstreetmap.org/8/127/85.png
```



Unfortunately, in a paper, you can't see interactive applications like this working. You can see it in action, and look at the code, at [8].

8. Mouse

Of course, what we *really* want is to be able to drag the map around with the mouse, not have to click on nudge buttons. Now we're really going to see the power of live data! We will want to know the position of the mouse, and the state of the button, up or down. So we create instance data for that:

```
<mouse>
  <x/><y/><state/>
</mouse>
```

and then we catch the mouse events:

```
<action ev:event="mousemove">
  <setvalue ref="mouse/x"
            value="event('clientX')"/>
  <setvalue ref="mouse/y"
            value="event('clientY')"/>
</action>
<action ev:event="mousedown">
  <setvalue ref="mouse/state">down</setvalue>
</action>
<action ev:event="mouseup">
  <setvalue ref="mouse/state">up</setvalue>
</action>
```

Now we have live data for the mouse!

We can show the state of the mouse by changing the mouse cursor from a hand into a clenched hand:

```
<div style="cursor: {
  if(mouse/state='up', 'pointer', 'move')
}">...
```

9. Capturing a move

The last bit is that we want is to save the start and end point of a move, so we can calculate how far we have dragged. The instance data is extended:

```
<mouse>
  <x/><y/><state/>
  <start><x/><y/></start>
  <end><x/><y/></end>
  <move><x/><y/></move>
</mouse>
```

We capture the start point of the drag when the mouse button goes down:

```
<action ev:event="mousedown">
  <setvalue ref="mouse/state">down</setvalue>
  <setvalue ref="mouse/start/x"
            value="event('clientX')"/>
  <setvalue ref="mouse/start/y"
            value="event('clientY')"/>
</action>
```

While the mouse button is down, we save the end position:

```
<bind ref="mouse/end/x"
      calculate="if(mouse/state = 'down',
                    mouse/x, .)"/>
<bind ref="mouse/end/y"
      calculate="if(mouse/state = 'down',
                    mouse/y, .)"/>
```

And calculate the distance moved as just end - start:

```
<bind ref="mouse/move/x"
      calculate="mouse/end/x - mouse/start/x"/>
<bind ref="mouse/move/y"
      calculate="mouse/end/y - mouse/start/y"/>
```

10. Dragging the map

So now we have the scaffolding we need to be able to drag the map. You may recall that the position of the map is recorded in `posx` and `posy`. That position now also depends on the mouse dragging. So we add instance data to record the last position:

```
<lastx/><lasty/>
```

and add a calculation to keep `posx` and `posy` updated (remember `scale` is the number of positions represented on a tile, so we divide by the tile size to get the number of positions represented by a pixel):

```
<bind ref="posx"
      calculate="../lastx -
                 ../mouse/move/x *
                 (../scale div ../tilesize)"/>
<bind ref="posy"
      calculate="../lasty -
                 ../mouse/move/y *
                 (../scale div ../tilesize)"/>
```

and only one other thing, namely reset `lastx` and `lasty` when the dragging stops:

```
<action ev:event="mouseup">
  <setvalue ref="lastx"
            value="posx"/>
  <setvalue ref="lasty"
            value="posy"/>
  <setvalue ref="mouse/start/x"
            value="mouse/end/x"/>
  <setvalue ref="mouse/start/y"
            value="mouse/end/y"/>
</action>
```

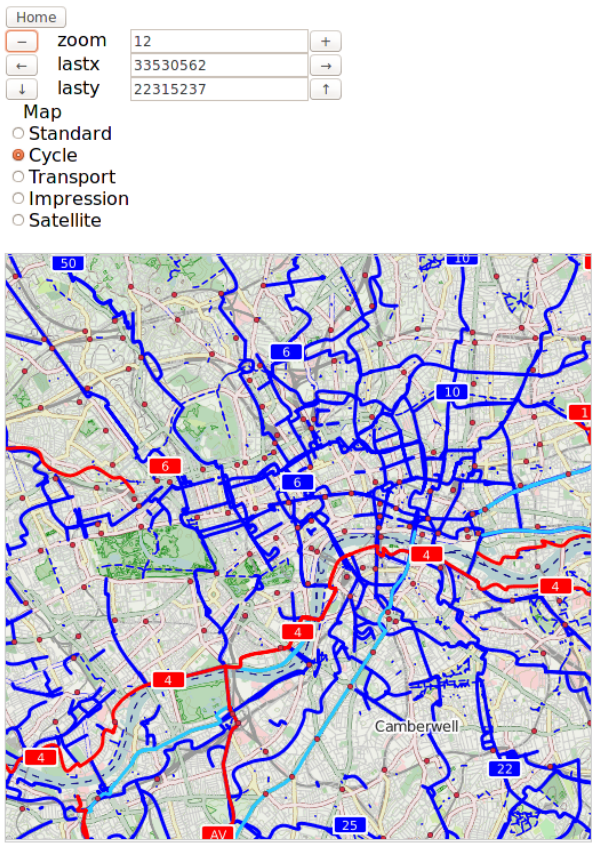
Now it is possible to drag the map around. Although from the user's point of view it feels like you are grabbing the map and dragging it around, all that is happening underneath is that we are tracking the live data representing the mouse, and using it to alter the live data that represents the centre of the map.

11. Bells. Whistles

Once we have this foundation, it is *trivial* to add things like a "Home" button, to add keystroke shortcuts, to zoom in and out with the mouse wheel, or to select tiles for another version of the map. For instance:

```
<select1 ref="site">
  <label>Map</label>
  <item>
    <label>Standard</label>
    <value>
      http://tile.openstreetmap.org/
    </value>
  </item>
  <item>
    <label>Cycle</label>
    <value>
      http://tile.opencyclemap.org/cycle/
    </value>
  </item>
  <item>
    <label>Transport</label>
    <value>
      http://tile2.opencyclemap.org/transport/
    </value>
  </item>
  ...
</select1>
```

Thanks to the live data, any time a different value is selected for "site", all the tiles get updated, without any further work from us.



Home

–	zoom	12	+
←	lastx	33530562	→
↓	lasty	22315237	↑

Map
- Standard
- [✓ Cycle]
- Transport
- Impression
- Satellite

12. Implementation

The implementation used in the online version of this application is XSLTForms [9], a client-side implementation of XForms that runs in all modern browsers, using a mixture of XSLT and Javascript. However, the code only uses standard XForms, and does not use any special facilities of this implementation. As long as the XForms implementation correctly catches the DOM mouse events used, the code should work in any implementation of XForms.

13. Conclusion

In a very abstract sense, a map like the one presented above can be seen as the presentation of two values, an x and y coordinate, overlaid with an input control to affect the values of x and y. The ability of XForms to abstract the data out of an application and make the data live via simple declarative invariants that keep related values up to date makes the construction of interactive applications extremely simple. The above example map application is around 150 lines of XForms code, in sharp comparison with the several thousand lines that a procedural programming language would need.

References

[1] *A Conversation with Tim Bray, Searching for ways to tame the world's vast stores of information.*. Jim Gray. ACM Queue. 20-25. 3. 1. February 2005. http://queue.acm.org/detail.cfm?id=1046941

[2] *WIKIPEDIA OpenSearch Test Form*. Alain Couthures. http://www.agencexml.com/xsltforms/wikipediasearch.xml

[3] *XForms 1.0*. Micah Dubinko, Leigh Klotz, Roland Merrick, and T V Raman. W3C. 2003. http://www.w3.org/TR/2003/REC-xforms-20031014/

[4] *XForms 1.1*. John Boyer. W3C. 2009. http://www.w3.org/TR/2009/REC-xforms-20091020/

[5] *XForms 2.0 (draft)*. John Boyer, Erik Bruchez, Leigh Klotz, Steven Pemberton, and Nick Van den Bleeken. W3C. 2014. https://www.w3.org/MarkUp/Forms/wiki/XForms_2.0

[6] *XForms 1.1 Quick Reference*. Steven Pemberton. W3C. 2010. http://www.w3.org/MarkUp/Forms/2010/xforms11-qr.html

[7] *XForms for HTML Authors*. Steven Pemberton. W3C. 2010. http://www.w3.org/MarkUp/Forms/2010/xforms11-for-html-authors

[8] http://www.cwi.nl/~steven/Talks/2014/xml-london

[9] *XSLTForms*. http://www.agencexml.com/xsltforms

A. Credit

Open Street Map data is © OpenStreetMap contributors , licensed as CC BY-SA.

Schematron - More useful than you'd thought

Philip Fennell

MarkLogic

<philip.fennell@gmail.com>

Abstract

The Schematron XML validation language has been around for about as long as XML and has been used extensively for validation tasks outside the gamut of what XML Schema 1.0 was designed for. The reference implementation is implemented, with great care, in XSLT, and with extensibility in mind. There are a number of points in the Schematron compilation process that provide opportunities to extend its basic behavior and allow other modes of report output to be generated. This paper looks at one example of extending Schematron to create an XML to RDF Mapping Language for flexible RDF triple construction and built-in source-data validation rules.

1. Introduction

This paper looks at one example of extending Schematron [1] to create an XML to RDF Mapping Language with built-in data validation rules and flexible RDF triple construction.

The Schematron XML validation language has been around for about as long as XML and has been used extensively for all those validation tasks that fell outside the gamut of what XML Schema 1.0 was designed for. Put simply, Schematron allows you to define a context for a set of rule tests that, when applied to a source XML document, must fail if they're not met or to report on the occurrences of nodes that you want to know about, and all using just XPath expressions.

```
<iso:schema
  xmlns:iso="http://purl.oclc.org/dsdl/schematron">

  <iso:ns prefix="atom"
          uri="http://www.w3.org/2005/Atom"/>
  <iso:title>Simple Atom Feed Rules</iso:title>

  <iso:pattern>
    <iso:title>Atom Feed Root</iso:title>
    <iso:rule context="/">
      <iso:assert test="atom:feed">
        The document root must be
        an atom:feed element.
      </iso:assert>
    </iso:rule>
  </iso:pattern>

  <iso:pattern>
    <iso:title>Required elements of
               an Atom Feed</iso:title>
    <iso:rule context="/atom:feed">
      <iso:assert test="atom:title">
        atom:title is missing,
        this is a required element.
      </iso:assert>
      <iso:assert test="atom:id">
        atom:id is missing,
        this is a required element.
      </iso:assert>
      <iso:assert test="atom:updated">
        atom:updated is missing,
        this is a required element.
      </iso:assert>
    </iso:rule>
  </iso:pattern>
</iso:schema>
```

In the above example two contexts are defined, one for the document root, so we can test for the presence of the root element and the second is the root `/atom:feed` element itself, so we can test for its required child nodes. It is this principle that we will use later to help define the contexts for mapping XML nodes to RDF triples.

Now, it's not that Schematron is under used and nor is it under appreciated but still there is a lot more to Schematron than meets the eye. Firstly, the reference implementation is *implemented*, with great care, in XSLT [2]. This must have seemed a natural choice given the type of input documents, XML, and the community of developers that it would be serving and also not withstanding the fact that, as XML, XSLT can be created and transformed by XSLT - quite possibly my most favourite aspect of XSLT.

The original core of the reference implementation was written with extensibility in mind but its output, the validation report, was somewhat limited by being a plain text format. Therefore, secondly, with its endorsement as an ISO standard, came the Schematron Validation and Reporting Language (SVRL), which gave the report a structured XML output. Finally, the multi-step *pipeline* that is used to compile the Schematron schema into XSLT allows for multiple extension points to suite your desired application.

2. Where else to use a rules-based reporting language?

With the increasing interest in the Semantic Web technologies, a lot of work has been done to bridge the gap between the existing RDBMS world and that of the Semantic Web so that the vast array of Relational systems can be queried with the SPARQL query language [3] as though they were actually RDF graphs. This requires either a simplistic direct mapping from table rows and columns to RDF triples or a more thoughtful, and in reality far more practical, one that uses an intermediate mapping language to describe the construction of triples from the table schema. The W3C have Recommendations for both a Direct Mapping (RDB2RDF) [4] and a Relational to RDF Mapping Language (R2RML) [5].

Tools currently exist that support both the Direct Mapping and the Mapping Language and can be used either as a SPARQL query layer, on top of an RDBMS, or as a standalone export tool that will dump a table, or tables, as RDF. As an aside, I think if you are to use these tools then you get the most value out of the SPARQL layer in prototyping the Mapping and then use the mapping to dump the database to RDF so you can bulk ingest into your Triple Store.

But what of XML to RDF? Certainly there is plenty of XML content, feeds, streams and data that exist out there but the options for extracting and mapping that information to RDF are confined to bespoke pieces of code. What would be useful is a way to express, in an XML and RDF oriented way, the identification of targets in the source XML, the context for a set of triples, define the subject URI, the properties (elements and attributes) you wish to map to your target RDF vocabulary and where to get the values that will become the objects of your triples. A rule-based language, like Schematron, gives us a hint as to how this might work.

3. A sketch of how we might use Schematron to map XML to RDF

If we look again at what a Schematron rule actually does, it defines a context, a node set, within an XML document, and allows you, by the use of XPath, to assert a validity test or report on the occurrence of a node relative to that context. Putting aside validity assertions for the moment, the `iso:report` instruction is the key here; although originally intended to simply report the presence of a node that matches a specific XPath pattern, it can be overridden to output, instead of a simple textual report message, any structured XML you require - in our case an RDF triple for each occurrence of the target node. The target node maps to the predicate of an RDF triple. Schematron has an `iso:value-of` instruction which, when used in conjunction with `iso:report`, can insert values, from the source document, into the message or in this case the *object* value of our triple. For each predicate/object pair one must define a subject URI, which can be derived wholly, or in part, from the context of the rule. More on this last aspect later.

Source XML:

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>Example Feed</title>
  <link href="http://example.org/"/>
  <updated>2003-12-13T18:30:02Z</updated>
  <id>http://example.org/feeds/60a76c80</id>
</feed>
```

Mapped RDF Triples:

```
@prefix rdfs:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:      <http://www.w3.org/2001/XMLSchema#> .
@prefix atom-owl: <http://bblfish.net/work/atom-owl/2006-06-06/#> .

<http://example.org/feeds/60a76c80> rdfs:type atom-owl:Feed ;
  atom-owl:title "Example Feed"@en ;
  atom-owl:updated "2003-12-13T18:30:02Z"^^xsd:dateTime .
```

In the above example we can see that there is a mapping that has been applied from the source Atom XML document to an RDF vocabulary called AtomOwl [6] which is an RDF modelling of Atom Feed format [7]. The following schema fragment illustrates how a conventional Schematron report can capture the necessary information to build the subject, predicate and object components of an RDF triple:

```
<iso:rule context="/atom:feed">
  <iso:assert test="atom:title">
    atom:title is missing,
    this is a required element.
  </iso:assert>
  <iso:report test="atom:title">
    The '<iso:value-of select="atom:id/text()"/>'
    feed has a title of
    '<iso:value-of select="atom:title/text()"/>'
    </iso:report>
</iso:rule>
```

The '**http://example.org/feeds/60a76c80**' feed has a **title** of '**Example Feed**'

In addition, if we want to introduce data validity constraints then the Schematron `iso:assert` instruction is still available for ensuring the integrity of the source XML before it is mapped to RDF.

4. XML Scissor-lift: An XML to RDF Mapping Language

May be it should have been called X2RML (XML to RDF Mapping Language), but, XML Scissor-lift derives its name from the term 'schema lifting', which is used by the Semantic Web community to describe the process of lifting the semantics of an XML grammar to a *higher* level by mapping its elements and attributes to terms in a target RDF vocabulary.

Scissor-lift uses Schematron along with aspects of the XML Pipeline Language (XProc) [8], URI Templates [9], the Web Application Description Language (WADL) [10] and the Triples in XML (TriX) [11] representation of RDF to provide some pre-existing patterns that should make it quicker to pick-up. XSLT is used for pre-compilation processing, extending Schematron and post mapping conversion to alternative RDF graph serializations.

4.1. A Basic Mapping Description

Looking at the previous source XML (Atom Feed) and the target RDF Vocabulary (AtomOWL), we see here a simple Scissor-lift mapping document:

```
<lift xmlns="https://github.com/anonymous/scissor-lift" xmlns:atom="http://www.w3.org/2005/Atom"
      name="atom-to-atom-owl" version="1.0">

  <title>Simple Atom Feed Rules</title>

  <variable name="feedURI" select="concat('http://example.org/feeds/',
                                    substring-after(/atom:feed/atom:id, 'feeds/'))"/>
  <pattern>
    <title>Atom Feed</title>
    <rule context="/">
      <triple match="atom:feed">
        <uri select="$feedURI"/>
        <uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
        <uri>http://bblfish.net/work/atom-owl/2006-06-06/#Feed</uri>
      </triple>
    </rule>
    <rule context="/atom:feed">
      <triple match="atom:title">
        <uri select="$feedURI"/>
        <uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</uri>
        <plainLiteral xml:lang="en" select="atom:title"/>
      </triple>
      <triple match="atom:updated">
        <uri select="$feedURI"/>
        <uri>http://bblfish.net/work/atom-owl/2006-06-06/#updated</uri>
        <typedLiteral
            datatype="http://www.w3.org/2001/XMLSchema#dateTime"
            select="atom:updated"/>
      </triple>
    </rule>
  </pattern>
</lift>
```

In the above example we first define a feedURI variable that takes its value from the feed's atom:id element, this variable has global scope so can be used in any mapping rule. Then we define a pattern for the Feed which contains two rules; the first rule's context is the document's root and we generate a triple on matching the atom:feed element where we map the element to the Feed class of the AtomOwl Ontology. The match attribute takes an XPath expression who's context is that of the containing rule. The second rule's context is the atom:feed element itself and we generate triples upon matching the atom:title and atom:updated elements. In both these cases we identify the predicates from the target vocabulary's title and updated terms respectively and selects the value of the objects from the atomized value of their matched elements.

The grammar for the triples is taken directly from the TriX RDF representation; the uri, plainLiteral and typedLiteral elements have been extended to accept either a literal string as their value or a select attribute which takes an XPath expression, including declared variables, that are evaluated at run time to provide the value for those elements. The context for the XPath expression is the context expression for the enclosing rule.

By default, Scissor-lift will generate a simple XML representation of the resulting graph using the TriX format, which directly corresponds to the format that we use to express the mapping:

```xml
<trix xmlns="http://www.w3.org/2004/03/trix/trix-1/">
  <graph>
    <uri/>
    <triple>
      <uri>http://example.org/feeds/60a76c80</uri>
      <uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#Feed</uri>
    </triple>
    <triple>
      <uri>http://example.org/feeds/60a76c80</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</uri>
      <plainLiteral xml:lang="en">Example Feed</plainLiteral>
    </triple>
    <triple>
      <uri>http://example.org/feeds/60a76c80</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#updated</uri>
      <typedLiteral
          datatype="http://www.w3.org/2001/XMLSchema#dateTime">
        2003-12-13T18:30:02Z
      </typedLiteral>
    </triple>
  </graph>
</trix>
```

From the TriX representation it is easy to generate N-Triples [12], Turtle [13], JSON-LD [14] and RDF/XML [15] using a subsequent XSLT transform.

4.2. Compiling and Executing a Mapping

As the mapping language is an extension of Schematron so its compiler implementation is an extension of the Schematron transforms. ISO Schematron uses a three step compilation process to generate the resulting XSLT transform:

1. Include
2. Expand
3. Compile

which Scissor-lift adds a 'Translate' step before Schematron's Include step. The translation step converts the `lift` document into a ISO Schematron schema document:

```
<rule context="/atom:feed">
  <triple match="atom:title">
    <uri select="$feedURI"/>
    <uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</uri>
    <plainLiteral xml:lang="en" select="atom:title"/>
  </triple>
</rule>
```

becomes:

```
<iso:rule context="/atom:feed">
  <iso:report test="atom:title">
    <sl:uri select="$feedURI"/>
    <sl:uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</sl:uri>
    <sl:plainLiteral xml:lang="en" select="atom:title"/>
  </iso:report>
</iso:rule>
```

The children of the `iso:reprot` element are not in the ISO Schematron namespace so are not touched by Schematron's Include and Expand transforms but are processed, as we will see later, by the extensions to the Compile transform.

The resulting XSLT transform can be used to convert individual XML source document instances into RDF and the transform can be run in editors like oXygen [16], where I tend to do all my development, and I use an XProc pipeline to orchestrate the compilation and transformation processes.

The transform can also operate in an XML database where it can be set to work converting XML documents in the database to RDF that can, in turn, be inserted into an RDF Triple Store. In the case of sending triples to a Triple Store, one of the standard RDF representations can be generated by using an XSLT transform to convert the afore mentioned TriX output into one of the recommended representations: N-Triples, Turtle, JSON-LD or RDF/XML.

4.3. Advanced Mapping Features

The following sections look at more advanced features of Scissor-lift that help us to build URIs, re-use definitions and determine data types.

4.3.1. Constructing URIs - URI Templates

A useful feature that was copied from WADL, but is a standard in its own right, are URI templates.

```
<pattern>
  <title>Atom Entry</title>

  <rule context="/atom:feed/atom:entry">
    <triple match=".">
      <uri template="http://example.org/feeds/{feedID}/entries/{entryID}">
        <param name="feedID"
            select="substring-after(/atom:feed/atom:id, 'feeds/')"
            type="xs:string"/>
        <param name="entryID"
            select="substring-after(atom:id, 'entries/')"
            type="xs:string"/>
      </uri>
      <uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#Entry</uri>
    </triple>
  </rule>
</pattern>
```

Rather than using, a potentially long-winded, `concat()` XPath expression to build a URI within a `select` attribute we can make use of the URI Template syntax and parameter declarations to insert values into the URI declared in the `template` attribute. In the above example we have the subject URI being comprised of two variable components, the feed's ID `{feedID}` and the entry's ID `{entryID}`. The `param` elements define the expressions that retrieve their respective values from the source XML. Although, not strictly speak required, the `type` attribute is provided to remind you what type of value you're generating.

The URI templates can be used on subject, predicate and object URIs, giving you maximum flexibility in create mappings.

4.3.2. Reuse - Abstract Patterns and Rules

Another feature that has been re-used from Schematron is the Abstract Patterns and Rules. Defining groups of triple mappings that can be re-used throughout the whole mapping saves a lot of time and duplication. In the following example we will see that there are a set of required elements for Atom that are common to both the feed and its entries: `title`, `id` and `updated`. The 'required' pattern is defined as abstract and its definition is used where a real pattern identifies itself 'is-a' instance of the 'required' pattern.

```
<pattern abstract="true" id="required">
  <title>Abstract Required</title>

  <rule context="$context">
    <triple match="atom:title">
      <uri select="$thisURI"/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</uri>
      <plainLiteral xml:lang="en-GB" select="atom:title"/>
    </triple>
    <triple match="atom:id">
      <uri select="$thisURI"/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#id</uri>
      <typedLiteral datatype="http://www.w3.org/2001/XMLSchema#anyURI"
          select="atom:id"/>
    </triple>
    <triple match="atom:updated">
      <uri select="$thisURI"/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#updated</uri>
      <typedLiteral datatype="http://www.w3.org/2001/XMLSchema#dateTime"
          select="atom:updated"/>
    </triple>
  </rule>
</pattern>

<pattern id="feed-required" is-a="required">
  <title>Feed Required</title>
  <param name="context" select="/atom:feed"/>
  <param name="thisURI" select="$feedURI"/>
</pattern>

<pattern id="entry-required" is-a="required">
  <title>Entry Required</title>
  <param name="context" select="/atom:feed/atom:entry"/>
  <param name="thisURI"
      select="concat($feedURI, '/entries/', substring-after(atom:id, 'entries/'))"/>
</pattern>
```

The parameters `context` and `thisURI`, defined where the abstract pattern is used, have their values substituted into the resulting triple mappings.

A more complex mapping used to convert an Atom `link` element into its corresponding triples can be defined as an abstract rule:

```
<pattern abstract="true">
  <title>Abstract Rules</title>

  <rule id="links" abstract="true">
    <triple match="." >
      <uri select="$feedURI"/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#link</uri>
      <id select="."/>
    </triple>
    <triple match="." >
      <id select="."/>
      <uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#Link</uri>
    </triple>
    <triple match="." >
      <id select="."/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#to</uri>
      <id select="@href"/>
    </triple>
    <triple match="@href" >
      <id select="@href"/>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#src</uri>
      <uri select="@href"/>
    </triple>
  </rule>
</pattern>

<pattern>
  <title>Atom Feed Root</title>

  <rule context="/">
    <triple match="atom:feed">
      <uri select="$feedURI"/>
      <uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</uri>
      <uri>http://bblfish.net/work/atom-owl/2006-06-06/#Feed</uri>
    </triple>
  </rule>

  <rule context="/atom:feed/atom:link">
    <extends rule="links"/>
  </rule>
</pattern>
```

and re-used in a pattern for the given context. Here the abstract rule is defined in the same way as the abstract pattern but is used where a rule declares that it 'extends' the abstract rule.

4.3.3. Hierarchical Structure and Blank Nodes

Quite often there will be situations where you need to map some XML node to a Class in the target vocabulary. In the previous example the `atom:link` element has attributes we also need to capture in the mapping, and although the link element is implicitly related to the feed by being a child of that feed we need to express that explicitly in RDF. The link has no specific ID (an anonymous resource) so we use the concept of a Blank Node (BNode) to assign a unique identifier to the link that is unique within the scope of the graph.

Scissor-lift provides a `select` attribute on the `id` element to define a context for creating the unique ID that can be shared by the respective subject and object URIs.

4.3.4. Type Discovery - XML Schema Reflection

As mentioned previously, the mapping process can run in an XML database, and in the case of MarkLogic [17] this enables an additional feature of Scissor-lift - auto-datatype discovery. If the source XML documents have an XML Schema associated with them, that is also loaded into MarkLogic, then during the conversion to RDF the transforms make use of MarkLogic's, little known, Schema Components API.

You can explicitly set the type of the triple's object using the `type` attribute on the `typedLiteral` element. However, if you leave this attribute off and, as already mentioned, you have an XML Schema in place, the transform will use the XML type annotations on the source XML to determine the base, Simple Type, of the source nodes and automatically assign that to their corresponding triple objects.

4.3.5. Data Integrity Checking - Assertions

Scissor-lift makes use of Schematron's `iso:report` instruction to carry the triple annotations into the compilation process. This leaves us free to use Schematron's `iso:assert` instruction to provide content validation tests that will help us ensure the integrity of the source XML.

```
<rule context="/atom:feed">
  <assert match="atom:title">atom:title is a required element.</assert>
  <triple match="atom:title">
    <uri select="$thisURI"/>
    <uri>http://bblfish.net/work/atom-owl/2006-06-06/#title</uri>
    <plainLiteral xml:lang="en-GB" select="atom:title"/>
  </triple>
</rule>
```

The above example illustrates how an assertion can be used as part of a triple mapping to ensure that the source XML contains a required element. The mapping process will fail for the context document if an assertion fails.

5. How Schematron was Extended

The reference implementation of Schematron, which uses XSLT for the compilation and validation execution, has been written with extension in mind. This was how the earlier version was extended to incorporate the SVRL report output. Starting from this point, it is relatively simple to override the SVRL generating templates to create the XML you wish to output from a report.

Scissor-lift uses XSLT's import mechanism to override the original behaviours of the `iso_schematron_skeleton_for_saxon.xsl` transform, where the main jump-off points are the "`process-root`" and "`process-report`" named templates which, as their names suggest, are the templates that processes the container for the rules and the `iso:report` instructions respectively. From this point onwards it is a case of creating templates that process the Scissor-lift triple-mapping instructions that will replace the original SVRL output.

(5. continued) Resulting from the compiler, the XSLT transform contains the templates that match the rule contexts and the logic that triggers the triple generation according to the matches declared in the mapping rules.

6. Other Mapping Options

There is another, pre-existing XML to RDF mapping technique that was devised as part of the all-encompassing W3C Web Services work.

6.1. Semantic Annotations for WSDL and XML Schema (SAWSDL)

The SAWSDL [18] technique uses annotation written into an XML Schema to define the mapping from XML to a target RDF vocabulary.

```
<xs:complexType name="feedType"
               sawsdl:modelReference=
"http://bblfish.net/work/atom-owl/2006-06-06/#Feed">
  ...
</xs:complexType>
```

The above example demonstrates a simple mapping of the feed's root element to AtomOwl's Feed class. This approach works fine for simple cases but when building a SAWSDL processor you are very limited as to how you create subject and object URIs without reverting to referencing external XSLT transforms to do the complicated pieces.

Other issues with SAWSDL include the need for an XML Schema in the first place and the willingness to add annotations to it. It was these restriction that provided the impetus to come up with an alternative approach that was both flexible and could be applied to arbitrary and schema-less XML.

7. Conclusions

Schematron as a rule-based XML validation language is very effective and used extensively to build bespoke validation rule sets. The way the XSLT reference implementation was built has made it relatively straight forward to extended and override its original behaviour to allow new instructions into the Schematron grammar to generate different outputs from a rule-set processed by the compiler.

XML Scissor-lift is one such extension of Schematron that illustrates how Schematron can be used to generate a completely different type of 'report' document. The pipelines, transforms and related code have been used in a number of projects to map existing XML source content/data to RDF to good effect.

8. Further Work

Any application that embeds XPath into its documents should be able to 'validate' the XPath expressions prior to evaluating them. Scissor-lift requires an additional pipeline step, before 'translation', to check that the XPath expressions are valid. This can be easily achieved using a user-defined XProc step to evaluate all the `select` and `match` attribute expressions against a 'dummy' source document. There is no need for the expressions to match anything but the simple act of attempting to evaluate them will allow the XProc engine to parse the expression and report an syntax errors.

The Scissor-lift project can be found on GitHub [19] where more documentation and examples are waiting to be created.

References

[1] *ISO Schematron*. 01 June 2006.
http://www.schematron.com/

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

[3] *SPARQL 1.1 Query Language*. W3C Recommendation. 21 March 2013.
http://www.w3.org/TR/sparql11-query/

[4] *A Direct Mapping of Relational Data to RDF*. W3C Recommendation. 27 September 2012.
http://www.w3.org/TR/rdb-direct-mapping/

[5] *RDB to RDF Mapping Language*. W3C Recommendation. 27 September 2012.
http://www.w3.org/TR/r2rml/

[6] *AtomOwl Vocabulary*. 26 June 2006.
http://bblfish.net/work/atom-owl/2006-06-06/AtomOwl.html

[7] *Atom Syndication Format*. December 2005.
http://tools.ietf.org/html/rfc4287

[8] *XML Pipeline Language (XProc)*. W3C Recommendation. 11 May 2010.
http://www.w3.org/TR/xproc/

[9] *URI Templates*. March 2012.
http://tools.ietf.org/html/rfc6570

[10] *Web Application Description Language (WADL)*. W3C Submission. 31 August 2009.
http://www.w3.org/Submission/wadl/

[11] *Triples in XML (TriX)*. 13 May 2004.
http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html

[12] *N-Triples*. W3C Recommendation. 25 February 2014.
http://www.w3.org/TR/n-triples/

[13] *Terse RDF Triple Language*. W3C Recommendation. 25 February 2014.
http://www.w3.org/TR/turtle/

[14] *A JSON-based Serialization for Linked Data*. W3C Recommendation. 16 January 2014.
http://www.w3.org/TR/json-ld/

[15] *RDF 1.1 XML Syntax*. W3C Recommendation. 25 February 2014.
http://www.w3.org/TR/rdf-syntax-grammar/

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

[17] MarkLogic Sever. by MarkLogic.
http://www.marklogic.com/

[18] *Semantic Annotations for WSDL and XML Schema*. W3C Recommendation. 28 August 2007.
http://www.w3.org/TR/sawsdl/

[19] *scissor-lift*. Philip Fennell.
https://github.com/philipfennell/scissor-lift

Linked Data in a .NET World

Kal Ahmed

Networked Planet Limited

1. Introduction

This paper discusses two different ways in which .NET applications can access linked data. We start with a discussion of using LINQ to query data from a SPARQL endpoint that will describe how and why you might use LINQ queries against a compile-time data model to query a dynamic, open data set. In the second section we discuss OData - Microsoft's approach to publishing data on the web - and its relationship to RDF and the Linked Data approach, and we show how an OData endpoint can be easily constructed as a type-safe "view" over an RDF data set.

2. LINQ to SPARQL

BrightstarDB[1] is an open-source triple store for .NET. Written entirely in C#, BrightstarDB builds on the DotNetRDF[2] library to provide a native .NET persistence mechanism for RDF complete with a RESTful API. However some of the coolest features of BrightstarDB are above the raw RDF/SPARQL level where we interact with the world of .NET. One of these is the way in which we provide runtime binding of an entity model to RDF triples and extend that to support converting .NET Language Integrated Query[3] (LINQ) queries to SPARQL queries.

2.1. Binding a Static Model to RDF

At first this probably seems like a mad idea. You have the flexibility with RDF to create and extend models; to work with data in an open-world model where anyone can add any properties to anything. A set of C# classes or interfaces are the exact opposite of that – a closed world, fixed at compile-time. Why would you want to give up on the freedom of RDF ?

Well there are a number of answers to that:

1. Domain models are useful abstractions. A fixed domain model provides focus and enables you to concentrate on expressing the solution to your problem(s). They provide a clear well-understood way to reason about the data that is being handled.
2. Compile-time models provide compile-time checking. This is not to say that dynamically typed programming languages aren't incredibly powerful, but there is something reassuring about catching errors before your IL gets generated.
3. A lot of useful .NET functionality (Intellisense for LINQ in particular) is based on introspection over the domain model.
4. In reality many applications have a data model that is constrained (e.g. by their user interface).
5. Actually you aren't giving anything up. The domain model does not constrain the underlying data it represents in anyway, it just provides a formalized way of dealing with one particular *view* of that data. Any number of different domain models can co-exist over a single data set, each providing access to different, possibly overlapping aspects of the data.

2.2. Data Binding Annotations

BrightstarDB uses a model-first approach to defining the objects that are to be data-bound to RDF. The programmer creates the interfaces that describes his domain model and uses attributes to decorate the interfaces with the additional information required to map types and properties to RDF resources. A default mapping is generated by convention, so the minimal amount of decoration required is simply a marker to tell the code-generator that this is an interface that needs to be processed.

```
[Entity]
public interface IPerson {
  string Id { get; }
  string Name {get; set;}
  ICollection<IPerson> Knows {get; set;}
}
```

[1] http://brightstardb.com/ and source code at https://github.com/BrightstarDB/BrightstarDB
[2] http://dotnetrdf.org/
[3] http://msdn.microsoft.com/en-us/library/bb397926.aspx

By default an interface is mapped to an RDF resource with a URI identifier generated from the default namespace URI with the name of the interface appended (with the leading 'I' removed). Similarly, properties are mapped to an RDF resource with a URI identifier generated from the default namespace URI plus the name of the property with the first character forced to lowercase. The default namespace can be customized using an assembly-scoped attribute.

The Id property holds the unique key value for the RDF resource that an instantiation of the interface will be bound to. This maps to a URI by appending the key to the default namespace URI. By convention this property may be called Id, ID or *`{Interface name without the leading 'I'}`*I[d|D]. So we could equally well use PersonId, ID or PersonID as our identifier property. Alternatively any other string property can be decorated with an [Identifier] attribute to indicate that it is the property to be used to reflect the entity key. This property is required to be read-only as the generated class will automatically assign IDs and provides a separate method for overriding the ID that ensures that the triple-store gets correctly updated.

The code generation writes a class that implements the interface. All of the properties included in the interface are implemented in the generated class. The data-binder allows all of the common C# value types including their nullable counterparts, Uri and ICollection<T> as long as the specified type is another interface that is decorated with the [Entity] attribute. Any methods declared on the interface are left unimplemented in the generated code, however the code generator produces a C# partial class which leaves the class open for the remainder of the interface to be implemented in a separate source file.

To map to specific URIs, some more decoration is required:

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IPerson {

  [Identifier("http://example.com/person/")]
  public string Id { get; }

  [PropertyType("http://xmlns.com/foaf/0.1/name")]
  public string Name { get; set; }

  [PropertyType("foaf:knows")]
  public ICollection<IPerson> Knows {get;set;}
}
```

The example above shows how we have now mapped the object to an existing RDF schema - in this case FOAF. The entity type resource URI can be specified in the [Entity] attribute; and to override the default property type resource URI for a property we add the [PropertyType] attribute. This example also shows that we can either use complete URIs or CURIEs. In the latter case, the CURIE prefix mapping is specified in an assembly-scoped attribute:

```
[assembly:NamespaceDeclaration(
  "foaf","http://xmlns.com/foaf/0.1/")]
```

In addition to data-binding the triples that the entity resource is the subject of, it is also possible to data-bind against the triples that the entity resource is the object of using the [InverseProperty] decorator:

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IPerson {

  [Identifier("http://example.com/person/")]
  public string Id { get; }

  [PropertyType("http://xmlns.com/foaf/0.1/name")]
  public string Name { get; set; }

  [PropertyType("foaf:knows")]
  public ICollection<IPerson> Knows {get;set;}
  [PropertyType(
    "http://example.com/foaf-ext/knownBy")]
  [InverseProperty("Knows")]
  public ICollection<IPerson> KnownBy { get; set; }
}
```

In the example above we add a KnownBy property which data-binds to the inverse of Knows - i.e. all the triples where the entity is the object of a `foaf:knows` triple. Note that the inverse binding is declared by specifying the property that data-binds the triple in its forward direction. Although in this case for simplicity the property is on the same interface, it is perfectly valid to name a property on another interface.

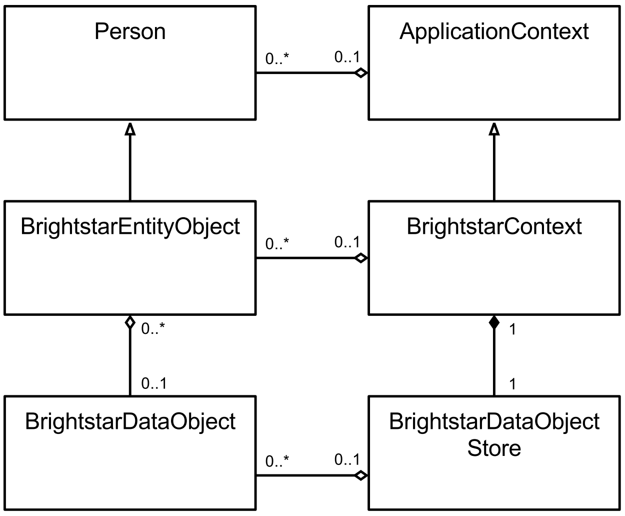
In the case that an RDF property is only to be data-bound in its reverse direction, it is possible to specify the property URI directly using an [InversePropertyType] attribute on the property:

```
[Entity("http://xmlns.com/foaf/0.1/Person")]
public interface IPerson {

  [Identifier("http://example.com/person/")]
  public string Id { get; }

  [PropertyType("http://xmlns.com/foaf/0.1/name")]
  public string Name { get; set; }

  // NOTE: no "Knows" property

  [InversePropertyType("foaf:knows")]
  public ICollection<IPerson> KnownBy { get; set; }
}
```

2.3. Data-Binding and Updates

The data-binding process itself is relatively straightforward and the model should be fairly familiar to anyone who has implemented or worked with an object-relational mapping tool. In addition to C# partial classes that implement the entity interfaces, the code-generation stage also generates an application context class that serves as the main entry point for client code. Both the generated entity classes and the generated application context class are derived from base classes that implement most of the mapping logic - the generated code is just a thin shim on top that implements the specific properties required by the domain model. These base classes themselves make use of the data-access layer provided by the BrightstarDataObjectStore, this handles the job of managing a collection of triples mapped to generic data objects. The BrightstarDataObjectStore also abstracts away the differences between BrightstarDB native stores and generic SPARQL endpoints.

Figure 1. Classes involved in the data-binding and update process



New entities are either created using their constructor and then added to the appropriate entity collection on the context class; or they are created and added to the context using a `Create` method provided on the entity collection of the context class. Existing entities are retrieved from the context object (typically via a LINQ query as discussed later). The data-binding can work either eagerly or lazily.

In the case of lazy loading, the initial query returns only a list of resource identifiers and the entities are instantiated with just the URI of the underlying RDF resource reflected as a string key value on the Identity property of the entity. When any other forward-direction property is accessed, a request is made to the triple-store to retrieve all of the triples in which the entity resource is the subject - in this way all properties other than inverse properties are populated in one round-trip. With inverse properties only the specifically mapped inverse properties are ever loaded. This pattern was chosen to enable the an entity to be designed in such a way as to avoid retrieving large inverse collections. For example when a small number of entities are used to classify a large collection of other entities (such as genres of movies), it is often desirable to avoid loading all of the movie-genre relationships when data-binding the genre entity.

In the case of eager loading, the LINQ query is converted to a SPARQL CONSTRUCT query which results in an RDF graph containing all the triples in which the entity resource is the subject. Inverse properties are always loaded lazily.

The data-binding process attempts to coerce RDF data-types to .NET types. Currently it handles a limited subset of the XML Schema datatypes, through a compile-time mapping.

When an existing resource is data-bound, the local state of the object is tracked in three parts - the quads loaded from the server, a collection of quads locally added but not yet sent to the server, and collection of deletion quad patterns held locally but not yet sent to the server. The difference between a quad and a quad pattern is that the latter allows a wildcard value to be used in one or more of the four parts of the quad - the wildcard matches all values so a quad containing wildcards may match multiple quads in the datastore. Depending on the operation carried out, items are added to or removed from one or both of these collections. The operations are summarized in the table below, where `E` is the URI of the entity resource, `PT` is the URI of the property, `V` is the new property value, `Vo` is the old property value, `C` is the update context, and `*` is the special quad pattern wildcard value.

The table below summarizes how these collections are updated under different circumstances.

Property Type	Operation	Modifications Made
Single-value forward property	Set initial value	`Add (E, PT, V, C) to AddTriples`

	Update value	`Add (E, PT, *, *) to DeletePatterns`
		`Add (E, PT, V, C) to AddTriples`
		`Remove all quads matching the pattern (E, PT, Vo, *) from AddTriples`

	Set value to null	`Add (E, PT, *, *) to DeletePatterns`
		`Remove all quads matching the pattern (E, PT, Vo, *) from AddTriples`

Collection forward property	Add item	`Add (E, PT, V, C) to AddTriples`

	Remove Item	`Add (E, PT, Vo, *) to DeletePatterns`
		`Remove all quads matching the pattern (E, PT, Vo. *) from AddTriples`

Inverse properties are handled slightly differently. If the property is mapped using the [InverseProperty] attribute, the update can be applied as an update to the entity that has the forward-direction property on it - this ensures that local modifications are properly reflected at both ends of the relationship. If the property is mapped to a URI identifier using the [InversePropertyType] attribute the update is applied by reversing `E` and `V` or `E` and `Vo` in the table above.

Local changes made to entities in the context are tracked in this way until the client application calls the `SaveChanges()` method on the context object. At this point a transaction is prepared and sent to the server. BrightstarDB supports two different types of update transaction. The first is native to the BrightstarDB store, in this transaction the set of triples to be added and triple patterns to be deleted are encoded as N-Triples and passed in a simple JSON data transfer object format. However, the code also supports connections to generic SPARQL UPDATE endpoints in which case the transaction is formatted as a SPARQL UPDATE request consisting of a DELETE WHERE clause to apply the delete patterns, followed by an INSERT DATA clause.

2.4. Type Mapping and Type Conversion

As we have already shown, each entity is mapped to an RDF resource which defines the entity type. This type is assigned to an entity instance using a standard `rdf:type` triple. However our goal is to support the flexibility to "view" an RDF resource as several different kinds of entity. Hence we have support for an RDF resource to concurrently map to more than one type of entity. All the mapped entities share the same underlying set of triples that are eagerly or lazily loaded from the server. The generated entity classes provide a `Become<T>()` method. `Become<T>()` is invoked to map the entity to a new entity type, this will add the required `rdf:type` triple locally and returns a new instance of T that is data-bound to the same set of underlying triples as the object that the `Become<T>()` method is invoked on. An parallel `Unbecome<T>()` method removes the `rdf:type` triple that maps the resource to the entity type T. `Become<T>()` and `Unbecome<T>()` provide a halfway-house between dynamic objects and static typing - an object can change its type dynamically at runtime but only between a small set of compile-time types.

2.5. Optimistic Locking

Our approach to data-binding also supports a simple form of optimistic locking when performing updates. The locking is based on applying a version number property on each resource. When the resource is loaded to access any of its properties, the version number property is retrieved and the version number is stored. If the resource currently has no version number, a new property is assigned with an initial version number of 1, but in this case the client will be unable to detect any concurrent server modifications.

On update, a conditional guard is added to the transaction that ensures that the version number quad still exists in the store, and a delete pattern and a new quad are added to increment the version number. With the BrightstarDB transaction format this guard is natively supported. The BrightstarDB transaction format allows guard statements that specify a collection of quads that must exist in the store, and a separate collection of quads that must not exist in the store prior to executing the transaction. These guards and the guarded updates are all processed in a single transaction, so either the guards pass and the update completes successfully or else no changes are made to the store. In the case that a guard fails, the failing quads are reported back to the client and these are used in the generated context class to propagate the errors in a meaningful way back to the client application. In the case of the BrightstarDB entity mapping, this means providing access to the list of entities that have been concurrently modified on the server.

Unfortunately this facility cannot be used when a store is updated using SPARQL UPDATE. The reason is that although SPARQL UPDATE supports conditional update through the use of DELETE WHERE and INSERT WHERE, the response from a SPARQL server does not distinguish between updates that completed without making modifications (because of failures to match in the WHERE clause) and updates that completed with the modifications applied. The lack of any report on (for example) number of triples modified makes it impossible to determine if the update has been applied successfully or not.

2.6. Mapping LINQ to SPARQL

Having a statically typed model for our domain enables us to make use of Language Integrated Query (LINQ). LINQ is a powerful set of features in .NET programming languages that allow programmers to write queries against SQL databases, XML documents, ADO.NET datasets and object collections using the same query language with full type-checking and Intellisense (auto-completion) support.

Typically LINQ is used in ORM scenarios to provide a bridge between a domain object oriented query language and SQL. However, LINQ can be used with other datastores by implementing a LINQ provider. BrightstarDB includes a LINQ provider with support for a significant subset of LINQ functionality that maps LINQ queries to SPARQL using the mapping metadata provided in the entity definitions. This functionality removes the need for developers to learn SPARQL in order to access data from a BrightstarDB or SPARQL endpoint.

The implementation in BrightstarDB is written to be completely SPARQL 1.1 compliant - it does not make use of any special features of BrightstarDB, and so it can be used with any SPARQL endpoint that supports version 1.1 of the Recommendation.

2.6.1. LINQ to SPARQL by Example

As already noted, the annotations used for data-binding enable us to construct a context object which exposes a collection for each type of entity. These collections are LINQ-queryable, and implement the logic required to convert a LINQ query on the collection into a SPARQL query. These collections are always the starting point for a LINQ query against the BrightstarDB entities.

Sample Data Model

In the following examples we will use these entity definitions:

```
[Entity]
public interface IDinner
{
  [Identifier("http://nerddinner.com/dinners/")]
  string Id { get; }
  string Title { get; set; }
  string Description { get; set; }
  DateTime EventDate { get; set; }
  string Address { get; set; }
  string City { get; set; }
  string HostedBy { get; set; }
  [PropertyType(
    "http://nerddinner.com/schema/attendees")]
  ICollection<IRSVP> RSVPs { get; set; }
}

[Entity]
public interface IRSVP
{
  [Identifier("http://nerddinner.com/rsvps/")]
  string Id { get; }
  string AttendeeEmail { get; set; }
  [InverseProperty("RSVPs")]
  IDinner Dinner { get; set; }
}
```

In the following discussion we will assume that the base URI configured for types is `http://nerddinner.com/schema/`. To make the SPARQL more readable, we will assume that the prefix string `nd` is mapped to `http://nerddinner.com/schema`. In practice the SPARQL query generator only ever uses full URIs in its generated queries.

Simple Selection and Traversal

The simplest query would be to return the ID of each instance of that type from the store. In LINQ:

```
from p in Context.Dinners select p.Id
```

We can convert this to a simple SPARQL query for all instances of a Dinner. The type URI convention tells us the URI to use for this type:

```
SELECT ?p WHERE {
  ?p a nd:Dinner .
}
```

Traversing a property is simply a question of adding another triple pattern to the query, so to get all RSVPs for a particular dinner, the following LINQ query could be written:

```
from p in Context.Dinners
where p.Id.Equals("1")
select p.Rsvps;
```

Note that in the LINQ query the complete URI identifier is not required, it is enough to specify just the entity key (the portion that follows the fixed base identifier URI). The RSVPs property of Dinner is mapped to the predicate type `http://nerddinner.com/schema/attendees`, so we can simply add a triple pattern using that predicate to the query:

```
SELECT ?v0 WHERE {
  <http://nerddinner.com/dinners/1>
            a nd:Dinner .
  <http://nerddinner.com/dinners/1>
            nd:attendees ?v0 .
}
```

This approach also works for relationships that run in the opposite direction to their mapped RDF predicate. So with this LINQ query:

```
from x in Context.Rsvps where
  x.Dinner.Id.Equals("1")
select x.Id
```

the property we are asked to traverse is the Dinner property of an IRsvp instance, which in our mapping is an inverse relationship, so the triple pattern we use appears "backwards":

```
SELECT ?x WHERE {
    ?x a nd:Rsvp .
    <http://nerddiner.com/dinners/1>
       nd:attendees ?x.
}
```

Selecting Property Values

Selecting individual properties from related entities is just a question of adding one more triple pattern:

```
from x in Context.Dinners
from r in x.Rsvps
select r.AttendeeEmail
```

Note also in this query we are finding all dinners and their attendees, not just a specific dinner.

```
SELECT ?v1 WHERE {
    ?x a nd:Dinner .
    ?r a nd:Rsvp .
    ?x nd:attendees ?r .
    ?r nd:email ?v1 .
}
```

Filters

Queries that filter on property values are usually easy to map to SPARQL FILTER (this example uses a different data model):

```
from x in Context.Dinners
where x.EventDate >= DateTime.UtcNow
select x.Id
```

Processing the LINQ query tree will reveal the datatype of the values in the query (in this case 1.3 is a System.Decimal. We provide a set of mappings between .NET datatypes and XML Schema datatypes.

```
SELECT ?x WHERE {
  ?x a nd:Dinner .
  ?x ns:eventDate ?v0 .
  FILTER (?v0 >=
    '2014-03-05T16:13:25Z'
    ^^<http://www.w3.org/2001/XMLSchema#dateTime>).
}
```

LINQ Method Calls

LINQ queries allow developers to incorporate the use of method calls within their query, obviously it is not possible to map all possible methods to SPARQL, but we can map some commonly used methods such as `Contains()` on a collection:

```
var cities = new string[] {
                "London", "Oxford", "Reading"};
var results = from x in Context.Dinners
              where cities.Contains(x.City)
              select x.Id;
```

This maps nicely to the use of IN in a FILTER:

```
SELECT ?x WHERE {
  ?x a nd:Dinner .
  ?x nd:city ?v0 .
  FILTER (?v0 IN ('London, 'Oxford', 'Reading')) .
}
```

Using `STRSTARTS()` and `STRENDS()` in SPARQL we can support `String.StartsWith()` and `String.EndsWith()` and using `REGEX()` we can support the variants that perform case-insensitive comparisons:

```
q = Context.Dinners.Where(
     c => c.Title.StartsWith("Bright",
              StringComparison.OrdinalIgnoreCase)
    ).Select(c => c.Id);
```

```
SELECT ?c WHERE {
  ?c a nd:Dinner .
  ?c nd:title ?v0 .
  FILTER (regex(?v0, '^Bright', 'i')).
}
```

Similarly we can use `CONTAINS()`, `STRLEN()`, `SUBSTR()`, `UCASE()` and `LCASE()` to support `String.Contains()`, `String.Length`, `String.Substring()`, `String.ToUpper()` and `String.ToLower()`. We can also use the various Date/Time functions to support the various properties of a .NET System.DateTime instance; and `ROUND()`, `FLOOR()` and `CEIL()` to support the equivalent functions from the .NET System.Math class.

Anonymous Objects

In addition to returning single properties and instances (more on which later), LINQ allows queries to return anonymous objects. These can be mapped quite easily to SPARQL queries that return multiple columns, at the cost of some extra client-side processing of the results set to generate the anonymous objects.

```
from x in Context.Dinners
select new {x.Title, x.EventDate};
```

In this case note the use of OPTIONAL to enable partially constructed anonymous objects:

```
SELECT ?v0 ?v1 WHERE {
  ?x a nd:Dinner .
  OPTIONAL { ?x nd:title ?v0 . }
  OPTIONAL { ?x nd:eventDate ?v1 .}
}
```

The properties returned in the anonymous object can also be the result of deeper traversal – the additional steps just get pushed into the OPTIONAL pattern.

```
from x in Context.Rsvps
select new {
  x.AttendeeEmail,
  DinnerTitle=x.Dinner.Title
};
```

Results in this generated SPARQL query:

```
SELECT ?v0 ?v2 WHERE {
  ?x a nd:Dinner .
  OPTIONAL { ?x nd:attendeeEmail ?v0 . }
  OPTIONAL {
    ?v1 nd:attendees ?x .
    ?v1 nd:title ?v2 .
  }
}
```

2.7. Eager Loading Complete Entities

The LINQ to SPARQL examples shown above typically return a single column when returning entities, and multiple columns when returning anonymous objects. In the former case, the returned column contains the URI of the entities that match the query criteria. This leads to a lazy-loading model that effectively requires N+1 server roundtrips to query for and retrieve N entities. In many cases, better performance will be achieved if the client is capable of retrieving all of the triples for the entities that match the query in a single request.

Note

In the following examples, the prefix string `bs` should be assumed to be mapped to the BrightstarDB-specific namespace URI `http://brightstardb.com/.well-known/model/`.

2.7.1. A Naive Approach

At first glance it seems like SPARQL 1.1 already provides us with all the necessary tools - CONSTRUCT and subqueries. We can put the generated SPARQL into a subquery, add a triple match pattern to expand the entity URI result into all triples where the entity URI is the subject and then use CONSTRUCT to generate the resulting graph.

```
CONSTRUCT {
  ?v0 ?v0_p ?v0_o .
} WHERE {
  ?v0 ?v0_p ?v0_o .
  SELECT ?v0 WHERE {
    ...
  }
}
```

On the client side we will receive an RDF graph. By grouping the triples by their distinct subject resource we can easily extract the result entities and all their triples to pass through to the data-binding layer.

This has the desired effect of returning all the triples we would normally load for the selected entities in a single query, and it is the basis of the approach we use to eager-load query results, but it does have some issues that need to be addressed.

2.7.2. Sorting

One place where the naive approach fails, is when it comes to retrieving sorted results. The problem is that the triples in the RDF graph we receive are not guaranteed to be in any particular order. Most of the time a SPARQL endpoint will return a graph in which the triples are in the logically expected order, but the SPARQL specification doesn't require this and so it is not a safe assumption to make.

In addition CONSTRUCT patterns can only contain variables projected from the WHERE clause and constants, so there is no way to insert a "position in the sort order" value into the graph built by a CONSTRUCT.

However when we are constructing the SPARQL, we do know which variables are to be used for sorting, the order (ascending vs descending) and the priority (sort by X then by Y). So we use this information to ensure that the sort values are added to the CONSTRUCTed graph using well-known predicates:

```
CONSTRUCT {
  ?v ?v0_p ?v0_o .
  ?v bs:sortValue0 ?sv0 .
  ?v bs:sortValue1 ?sv1 .
} WHERE {
  ?v0 ?v0_p ?v0_o .
  SELECT ?v0 ?sv0 ?sv1 WHERE {
  ...
  }
}
```

This generates a result graph in which each distinct subject resource has not only the triples for all of its properties, but also the sort values for ordering the results on the client. On the client we parse the results graph and execute a SPARQL query against that graph to select the entities URIs in their sort order:

```
SELECT ?x WHERE {
  ?x bs:sortValue0 ?sv0 .
  ?x bs:sortValue1 ?sv1 .
} ORDER BY ?sv0, DESC(?sv1)
```

2.7.3. Paging

If we don't use OFFSET and LIMIT then it is only necessary to sort on the client-side. However, when paging comes into play it is necessary to apply the sorting on the server-side to ensure that the correct slice of results gets returned. So the SPARQL sent to the server would include ORDER BY, OFFSET and LIMIT clauses:

```
CONSTRUCT {
  ?v ?v0_p ?v0_o .
  ?v bs:sortValue0 ?sv0 .
  ?v bs:sortValue1 ?sv1 .
} WHERE {
  ?v0 ?v0_p ?v0_o .
  SELECT ?v0 ?sv0 ?sv1 WHERE {
  ...
  } ORDER BY ?sv0 DESC(?sv1) OFFSET 10 LIMIT 10
}
```

Note that in this case we still need to apply the sorting client-side because there is no guarantee that the triples in the CONSTRUCTed graph are in sort order, but this sorting will be applied only to the single page of results returned by the server and as the server has already limited the results returned, we will not need to re-apply the paging.

2.7.4. Distinct Results

The `Distinct()` LINQ operator (or DISTINCT SPARQL keyword) adds another bit of complexity. Up to this point our result graphs are generated by expanding on the results returned by the subquery. If the query has no sorting applied, it is possible to simply apply the DISTINCT keyword to the subquery. However once the subquery is extended to project out the sort values, it can potentially end up returning the same entity binding multiple times.

To handle this the subquery needs to be refined. We refine the subquery to return the highest possible value for sort variables that are sorted in ascending order and the lowest possible value for sort variables that are sorted in descending order. This is achieved through the use of grouping and the MIN and MAX aggregates. By grouping by the entity URI and then using MAX and MIN aggregates on the sort values we ensure that the subquery returns only a single row for each binding yielding the values that sort highest in the final results:

```
CONSTRUCT {
  ?v ?v_p ?v_o .
  ?v bs:sortValue0 ?sv0 .
  ?v bs:sortValue1 ?sv1 .
} WHERE {
  ?v ?v_p ?v_o .
  {
    SELECT DISTINCT
      ?v
      (MAX(?v_sort0) AS ?sv0)
      (MIN(?v_sort1) as ?sv1)
    WHERE {
      ...
    }
    GROUP BY ?v
    ORDER BY ASC(MAX(?v_sort0)) DESC(MIN(?v_sort1))
  }
```

3. OData/SPARQL

This section is based on a position paper originally written for the W3C Open Data on the Web workshop[1].

OData[2] is a data access protocol designed to provide standard CRUD and query operations on a data source via HTTP interactions. OData is built on the AtomPub protocol using an Atom structure as the envelope that contains the data returned from each OData request.

We have been working on exposing SPARQL endpoints as read-only OData endpoints[3]. The following is a description of our motivation and some of the technical details about how we are approaching the problem.

At some level there is no doubt that the SPARQL, RDF, Linked Data Platform[4] stack "competes" with the OData stack. Both provide access to generic data models, both attempt to play well as a RESTFul architecture, both offer up query languages for use by remote clients over HTTP.

OData does a better job of exposing data as entities and OData support for containers and entity level update is more mature than the LDP effort. RDF on the other hand has a meta-model that is more accessible; the unwavering use of URIs for addressing and identification, both at the level of instance data and the level of ontology is a real benefit as is the inherent ability to merge heterogenous data.

However, "competes" is in scare-quotes because it is not useful to think of public open data standards competing with one another with each trying to operate to the exclusion of the other. Rather we should be focussing on the ways in which interoperability can be achieved as running code but also at the level of the standards making process. We have been working with both OData and RDF now for many years and trying to help bridge the gap between these web data worlds at the level of running code. Based on that experience, we have started a project to provide an OData endpoint that sits on top of any SPARQL-compliant endpoint. Technically this is a cool thing to play with, but it is also a way to get a feel for issues in open data protocol interoperability that could benefit from further standards work.

3.1. The Approach

The technical approach we have taken is to implement a proxy that parses OData operations and rewrites them as equivalent SPARQL operations that can then be executed against a SPARQL endpoint. The SPARQL result set is then parsed and rewritten as an OData result set.

The main hurdle to overcome is that an OData service is driven by the underlying domain model. An OData service exposes its ontology and entity collections as a metadata document[5] with a well-known URI[6], and all queries are expressed in terms of this model. For a generic SPARQL endpoint, this is not necessarily the case.

As the service we are exposing is OData we have chosen to approach this problem by making use of the annotations extension point in the OData service metadata document. By taking this approach we enable the OData service that the proxy provides to be defined either:

1. As a manually configured OData service metadata document.
2. By conversion from a known RDF schema or OWL ontology.
3. By introspection of the SPARQL endpoint using SPARQL queries that make use of RDF Schema / OWL types and properties.

At present we have not implemented either (2) or (3) but we are fairly confident that some level of useable OData service metadata could be automatically generated in this way. The other potential advantage is that the OData service metadata is simply another resource that can be published, so it would be possible for the owner of a SPARQL endpoint to make an official OData service description available even if they were unwilling/unable to host the OData proxy themselves.

3.2. OData Annotations

To get things working we created a small set of annotations that can be used to decorate the model described in an OData service metadata document. The annotations are used to help map OData entity and property types to their equivalent RDF types.

[1] http://www.w3.org/2013/04/odw/
[2] http://odata.org/
[3] https://github.com/BrightstarDB/odatasparql
[4] http://www.w3.org/2012/ldp
[5] http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part3-csdl.html
[6] http://docs.oasis-open.org/odata/odata/v4.0/os/part2-url-conventions/odata-v4.0-os-part2-url-conventions.html#_Toc372793772

We use the following annotation namespace for all SPARQL OData annotations:

```
<Using Namespace="ODataSparqlLib.Annotations"
       Alias="Sparql"/>
```

This is just a way of saying that the Annotation `ODataSparqlLib.Annotations.Uri` can be referenced in the metadata document using the short name `Sparql.Uri`.

3.2.1. Identity Prefix Annotation

The Identity Prefix Annotation is used to help map simple between RDF Resource URIs to OData simple identity attributes. For example in RDF we want to be using a URI such as `http://www.brightstardb.com/products/1` where as in the OData entity we want to talk about this as product with Id of '1', and connected to this as `/products(1)`. While it would be possible to use the full URI of an RDF resource as its OData identifier, it has implications for consistent URI escaping and for the length of the resulting OData URIs.

To achieve this we use the `IdentifierPrefix` annotation on the property identified as the `Key` property for the OData `EntityType`. The following sample show its usage.

```
<EntityType Name="Film">
  <Key>
    <PropertyRef Name="Id"/>
  </Key>
  <Property
      Name="Id" Type="Edm.String"
      Nullable="false">
    <ValueAnnotation
        Term="Sparql.IdentifierPrefix"
        String="http://dbpedia.org/resource/"/>
  </Property>
</EntityType>
```

In the above example an RDF resource with a URI like `http://dbpedia.org/resource/Un_Chien_Andalou` can be referenced through the OData proxy as
`http://example.org/odata/Films('Un_Chien_Andalou')`.

The identity prefix mapping is specific to an OData entity type, so each type can use a different prefix if required.

3.2.2. Entity Type Mapping Annotation

To indicate how types in OData map to types in RDF we use a `Uri` annotation on the OData `EntityType` definition. In this context the `Uri` annotation simply provides the full URI of the RDF resource that defines the entity type. The following sample shows its usage (the URI is truncated for readability):

```
<EntityType Name="Person">
  ...
  <ValueAnnotation
    Term="Sparql.Uri"
    String="http://mappings.dbpedia.org/…/Person"/>
</EntityType>
```

We also allow a default namespace URI for entity types to be defined in the proxy configuration file, any `EntityType` without an explicit mapping receives a mapping based on appending the `EntityType` name to the namespace URI. We allow for different string case mappings to also be applied when resolving the name to a URI e.g. force to lower-/upper-case; force to lower/upper camel-case. This can make for a very lean set of annotations on the OData model.

3.2.3. Literal Property Type Annotation

A similar approach is used to map OData properties to RDF properties.

```
<Property Name="Name" Type="Edm.String"
          Nullable="true">
  <ValueAnnotation
     Term="Sparql.Uri"
     String="http://dbpedia.org/property/name"/>
</Property>
```

Again, we also allow a default namespace URI for property types to be defined in the proxy configuration file (separate from the default namespace for entity types), any `Property` without an explicit mapping receives a mapping based on appending the Property name (with case conversion applied) to the namespace URI.

3.2.4. Association Property Type Annotations

In OData, properties that reference other entities are described by a NavigationProperty that defines a traversal of a separately defined Association type. This allows OData to support bidirectional traversal of the relationships between entities. In RDF resource to resource relationships are directed. To accommodate OData's bidirectionality we introduce an additional IsInverse annotation which can be used in conjunction with a Uri annotation to specify both the RDF property type and the direction in which the property is traversed (subject-to-object or object-to-subject).

```
<EntityType Name="Place">
  <NavigationProperty
       Name="BirthPlaceOf"
       Relationship="DBPedia.Person_BirthPlace"
       FromRole="BirthPlace"
       ToRole="Person">
    <ValueAnnotation
        Term="Sparql.Uri"
        String="http://dbpedia.org/…/birthPlace"/>
    <ValueAnnotation
        Term="Sparql.IsInverse"
        Boolean="True" />
  </NavigationProperty>
</EntityType>
<Association Name="Person_DeathPlace">
  <End Role="Person"
       Type="DBPedia.Person" Multiplicity="*"/>
  <End Role="DeathPlace"
       Type="DBPedia.Place" Multiplicity="1"/>
</Association>
```

Note that the Association definition is currently un-annotated as all the necessary information is conveyed by the annotations on the NavigationProperty.

Once again, the NavigationProperty Name can be combined with the default ontology base URI specified in the proxy configuration to avoid the need to provide explicit Uri annotations.

3.2.5. A Larger Example

Putting this all together here is a complete OData service metadata document with annotations that we can use to expose a subset of DBPedia as OData.

In this example, a base type namespace URI of http://dbpedia.org/ontology/ and a base property namespace of http://dbpedia.org/property/ is used and names are mapped to URI components by forcing them to lower camelcase. In this example, this means that many properties and entity types do not require a Uri annotation to be mapped.

```
<?xml version="1.0" encoding="utf8"?>
<edmx:Edmx xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx"
           Version="3.0">
  <edmx:DataServices
          xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
          m:DataServiceVersion="2.0">
    <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"
            Namespace="DBPedia">
      <Using Namespace="ODataSparqlLib.Annotations" Alias="Sparql"/>

      <!-- Thing:http://www.w3.org/2002/07/owl#Thing -->
      <EntityType Name="Thing">
        <Key>
          <PropertyRef Name="Id"/>
        </Key>
        <Property Name="Id" Type="Edm.String" Nullable="false">
          <ValueAnnotation Term="Sparql.IdentifierPrefix"
                           String="http://dbpedia.org/resource/"/>
        </Property>
        <ValueAnnotation Term="Sparql.Uri"
                         String="http://www.w3.org/2002/07/owl#Thing"/>
      </EntityType>

      <!-- Work: http://dbpedia.org/ontology/Work -->
      <EntityType Name="Work" BaseType="DBPedia.Thing">
        <!--Title: Gets default URI mapping: http://dbpedia.org/property/title -->
        <Property Name="Title" Type="Edm.String" Nullable="true"/>
        <!--Director: Gets default URI mapping:
                      http://dbpedia.org/property/director -->
        <NavigationProperty Name="Director"
                            Relationship="DBPedia.Work_Director"
                            FromRole="Work"
                            ToRole ="Director"/>
      </EntityType>

      <!-- Film: http://dbpedia.org/ontology/Film
                 Derived from Work -->
      <EntityType Name="Film" BaseType="DBPedia.Work">
        <Property Name="Name" Type="Edm.String" Nullable="true">

          <!--Not strictly necessary, but shown as an example -->
          <ValueAnnotation Term="Sparql.Uri"
                           String="http://dbpedia.org/property/name"/>
        </Property>
        <Property Name="Runtime" Type="Decimal" Nullable="true" />
        <Property Name="ImdbId" Type="Edm.String" Nullable="true"/>

        <!-- Not strictly necessary, but shown as an example -->
        <ValueAnnotation Term="Sparql.Uri"
                        String="http://dbpedia.org/ontology/Film" />
      </EntityType>

      <!-- http://dbpedia.org/ontology/Person -->
      <EntityType Name="Person" BaseType="DBPedia.Thing">
        <Property Name="Name" Type="Edm.String" Nullable="true">

          <!-- Here we use FOAF vocab for a property -->
          <ValueAnnotation Term="Sparql.Uri"
                           String="http://xmlns.com/foaf/0.1/name"/>
        </Property>
        <Property Name="BirthDate" Type="Edm.DateTimeOffset" Nullable="true">
          <ValueAnnotation Term="Sparql.Uri"
                           String="http://dbpedia.org/ontology/birthDate"/>
        </Property>
        <Property Name="DeathDate" Type="Edm.DateTimeOffset" Nullable="true">
          <ValueAnnotation Term="Sparql.Uri"
                           String="http://dbpedia.org/ontology/deathDate"/>
        </Property>
        <NavigationProperty Name="BirthPlace"
                            Relationship="DBPedia.Person_BirthPlace"
                            FromRole="Person"
                            ToRole="BirthPlace"/>
        <NavigationProperty Name="DeathPlace"
                            Relationship="DBPedia.Person_DeathPlace"
                            FromRole="Person"
                            ToRole="DeathPlace"/>
```

```
        <NavigationProperty Name="RestingPlace"
                            Relationship="DBPedia.Person_RestingPlace"
                            FromRole="Person" ToRole="RestingPlace"/>
      </EntityType>

      <!-- Place: http://dbpedia.org/ontology/Place -->
      <EntityType Name="Place" BaseType ="DBPedia.Thing">
        <Property Name="Abbreviation" Type="Edm.String" Nullable="true"/>
        <Property Name="Abstract" Type="Edm.String" Nullable="true"/>
        <Property Name="AnnualTemperature" Type="Edm.Decimal" Nullable="true"/>
        <Property Name="Elevation" Type="Edm.Decimal" Nullable="true"/>
        <Property Name="PopulationTotal" Type="Edm.Int32" Nullable="true"/>
      </EntityType>

      <Association Name="Work_Director">
        <End Role="Work" Type="DBPedia.Work" Multiplicity="*"/>
        <End Role="Director" Type="DBPedia.Person" Multiplicity="1"/>
      </Association>

      <Association Name="Person_BirthPlace">
        <End Role="Person" Type="DBPedia.Person" Multiplicity="*"/>
        <End Role="BirthPlace" Type="DBPedia.Place" Multiplicity="1"/>
      </Association>

      <Association Name="Person_DeathPlace">
        <End Role="Person" Type="DBPedia.Person" Multiplicity="*"/>
        <End Role="DeathPlace" Type="DBPedia.Place" Multiplicity="1"/>
      </Association>

      <Association Name="Person_RestingPlace">
        <End Role="Person" Type="DBPedia.Person" Multiplicity="*"/>
        <End Role="RestingPlace" Type="DBPedia.Place" Multiplicity="1"/>
      </Association>

      <EntityContainer Name ="Contents" m:IsDefaultEntityContainer ="true">
        <EntitySet Name="Films" EntityType="DBPedia.Film"/>
        <EntitySet Name="Persons" EntityType="DBPedia.Person"/>
        <EntitySet Name="Places" EntityType="DBPedia.Place"/>
        <AssociationSet Name="Film_Director" Association="DBPedia.Film_Director">
          <End Role="Film" EntitySet="Films"/>
          <End Role="Director" EntitySet="Persons"/>
        </AssociationSet>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

3.3. Request Transforms

Typical OData requests are for either a single entity, a set of entities or a select format that retrieves a result that appears as a table. To map this into a SPARQL query we use the above annotations and then depending on the target result use either the SELECT or CONSTRUCT keywords.

OData select queries work in a similar fashion to SELECT in SPARQL by returning a tabular result. So we generate a SPARQL SELECT query and map the resulting SPARQL table to an OData results set.

When processing OData queries that result in an entity or collection of entities, we use SPARQL CONSTRUCT queries. This allows us to retrieve an RDF graph from the SPARQL endpoint that represents the entities, their properties and relationships. The graph is then processed by the proxy into a list of entities and, if necessary the list is sorted by the sort criteria specified in the original OData query. The way the CONSTRUCT queries are built is very similar to the approach used to map LINQ queries to SPARQL as we have the same constraints and the same requirements around the return results.

3.4. Update

For now update is out of scope for this project, but it could potentially be implemented using the same set of annotations we use for read. The main difference is that we need to make use of SPARQL UPDATE, which is a separate specification and more importantly usually lives in a different place on the server, most likely, behind the firewall.

We see the initial benefit of this work as opening up existing RDF triple stores with public SPARQL endpoints to OData applications. Very few of these, for obvious reasons, offer a writeable SPARQL update endpoint. For enterprises with RDF stores the additional update option may be of interest.

3.5. Future work

In its current state, the library is best described as a minimal proof of concept with basic support for selecting entities by their ID or with a few simple property operators such as Equals and GreaterThan. We also have some basic sorting implemented. The current implementation is based on Microsoft OData libraries that implement v3 of the OData specification.

The main work to do is to create a complete implementation of all OData path semantics, filters options, and projection operators and upgrade the project to support OData v4. We would also like to provide tools for generating an OData service metadata document from a SPARQL endpoint or its ontology and to generate / write mappings for some of the existing public SPARQL endpoints.

Frameless for XML - The Reactive Revolution

Robbert Broersma

Frameless

`<robbert@frameless.io>`

Yolijn van der Kolk

Frameless

`<yolijn@frameless.io>`

Abstract

What would the web look like with functional reactive templates driven by functional reactive query expressions?

Lots of recent innovative developments are significant steps towards faster and more manageable web development, but to really improve our lives by leaps and bounds we must take a step back and consider the requirements for unleashing all this power to front-end developers that aren't fluent in JavaScript.

What would happen if we throw Angular expressions, React's virtual DOM and Reactive Extensions (Rx) in a mix? What if we use a declarative syntaxes like XSLT and XPath to compile an instruction set for this engine? What if we can reason about the instructions that make up your website and automatically build minimal and optimized modules?

It's uneconomical to obtain optimal performance for most projects you're working on, there are just too many sides to it: asynchronous tasks, web workers, parallel computations, lazily loading modules, reducing file size, splitting HTML/CSS/JS into modules, combining modules again to reduce HTTP requests, minification, atomic DOM updates, only rendering what's visible, only calculating what is being rendered, only re-calculating what has changed...

But we must do better, also because performance is very much about economic inclusiveness. Smaller web pages are essential to those using internet in (remote) areas over slow 2.5G mobile networks, where wireless data charges are high and every CPU cycle counts when you're using a $25 dollar smartphone.

When we've got a reactive template solution in place we can start thinking about using some of the kilobytes we've saved and some of the CPU cycles to add ubiquitous support for unsexy inclusive technologies such as accessibility, Unicode, localization, and security.

1. Introduction

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

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

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

Frameless is created to simplify application development and is, due to its API, great for writing readable code.

Frameless is built on our own XSLT processor running in the browser. It includes a custom built reactive XPath query engine for simple, powerful querying that works across browsers. This way we are able to include useful features like `$variables` and allowing custom functions in XPath. The current beta release works in all modern browsers, but also works in hostile environments such as IE6 and Firefox 1.0.

We achieve resilience in platform support by employing extensive feature detection and never correlating the presence of a certain platform feature to a certain browser version. This way Frameless releases keep working when browsers with new features are introduced, and it will fallback to more basic platform APIs when methods or properties are removed or renamed.

doi:10.14337/XMLLondon14.Broersma01

Developers can quickly get up to speed with our template engine, because we provide all template instructions directly inside HTML pages, using HTML syntax. These templates consist of custom elements and attributes that instruct Frameless to conditionally show or repeat markup snippets. In addition to this we also support value templates inside `{` and `}` brackets for both attributes and text content.

2. Why we created Frameless

Three years ago we started the development of Frameless, because we felt the development of complex web applications really lacks a solid basis. Browser differences, internationalization, scalability and security are recurring issues we think can and should be solved structurally and invisibly.

We wanted to be able to use templates directly in the HTML pages, rather then only in external files. While there is nothing wrong with external XSLT templates, we think there are some interesting advantages in being able to use them inline. When using a more basic syntax for XSLT, programming the templates will be more intuitive to front-end developers, while the resulting code is more readable and self explanatory than JavaScript would be.

XSLT and XPath from versions 2 and up provide very competitive feature sets, and are arguably more mature than the alternatives JavaScript libraries offer. That's why we chose to start development by implementing XSLT 2: designing for complex use cases can lead to a cleaner architecture than optimizing for simple use cases and later bolting on complexity.

By making the templates automatically react to changes in the data and to changing variables, implementing real-time user interfaces doesn't require writing extensive DOM-manipulation code anymore, drastically reducing development time and bugs.

Our mission is to create a framework that allows us to write web applications by functionally describing their flow and behavior, rather than describing what each browser needs to hear. We want to rely on the template renderer to resolve cross site scripting vulnerabilities and optimally render changes to the DOM.

3. Example: reactive filtering using full text search

Many user interfaces include search and autocomplete functionality, showing the results as-you-type. Using Frameless no additional code at all is needed to show these results in real time.

In the following example we'll implement finding a person from a list of contacts, where the code in Figure 1 produces the component found in Figure 2.

First there is a form `<input>` that is bound to the `$query` variable. As the user is typing text into this form field, the `$query` becomes more specific and filters out all non-matching contacts. For all contacts that do match, we want to highlight the matching parts of that person's name.

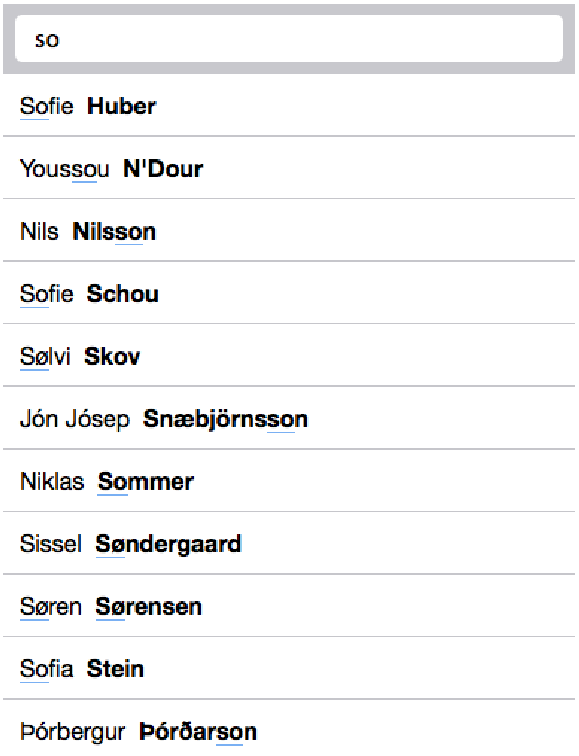
Unicode decomposition of text is helpful in separating the diacritics from the letters, allowing comparison against the search query without taking account diacritics. However, not all characters in Latin script can be decomposed to A-Z. That's why we've added a "confusables" option, that uses the Unicode database to also match characters that look like the original Latin letters. This way you can find names such as "Sørensen" and "Đặng", names that normally wouldn't even show up in search results.

To provide more than just regular expression based string templates, we allow custom tokenizers for the `<analyze-string>` instruction. These custom templates could for example be used to implement syntax highlighting for code in documents like this very document, but in this case we're using the `full-text` tokenizer to highlight matches to the search query.

Figure 1. Contact Search Code

```
<div class='contacts-app app'>
  <div class='contacts-finder data-list-container'>
    <div class='search data-list-search'>
      <input type='text' placeholder='Search' ref='$query' id='contact-query'>
    </div>
    <div id='contact-list' class='data-list' tabindex='0'>
      <for-each select='$contacts[name contains text $query using diacritics insensitive using case
                        insensitive using option confusables]'
                sort='(name/lastname, name/firstname)[1]'>
        <div class='data-list-item' tabindex='0' focus-action='open-contact(.)'>
          <analyze-string select='name/firstname' match='{$query}' tokenizer='full-text'>
            <matching-substring>
              <em>{{.}}</em>
            </matching-substring>
            <non-matching-substring>{{.}}</non-matching-substring>
          </analyze-string>
          <b>
            <analyze-string select='name/lastname' match='{$query}' tokenizer='full-text'>
              <matching-substring>
                <em>{{.}}</em>
              </matching-substring>
              <non-matching-substring>{{.}}</non-matching-substring>
            </analyze-string>
          </b>
        </div>
      </for-each>
    </div>
  </div>
</div>
```

Figure 2. Contact Search Component



so

Sofie **Huber**

Youssou **N'Dour**

Nils **Nilsson**

Sofie **Schou**

Sølvi **Skov**

Jón Jósep **Snæbjörnsson**

Niklas **Sommer**

Sissel **Søndergaard**

Søren **Sørensen**

Sofia **Stein**

Þórbergur **Þórðarson**

4. Example: reactive grouping and sorting for interactive infographics

In this code sample we will write a couple of lines of code found in Figure 3 to create an overview of the tallest structures in the world, visually comparing their sizes to other tall structures in that country. First there is an XPath query that loads an external file, containing the data. Once the file is loaded, the rest of the query is executed.

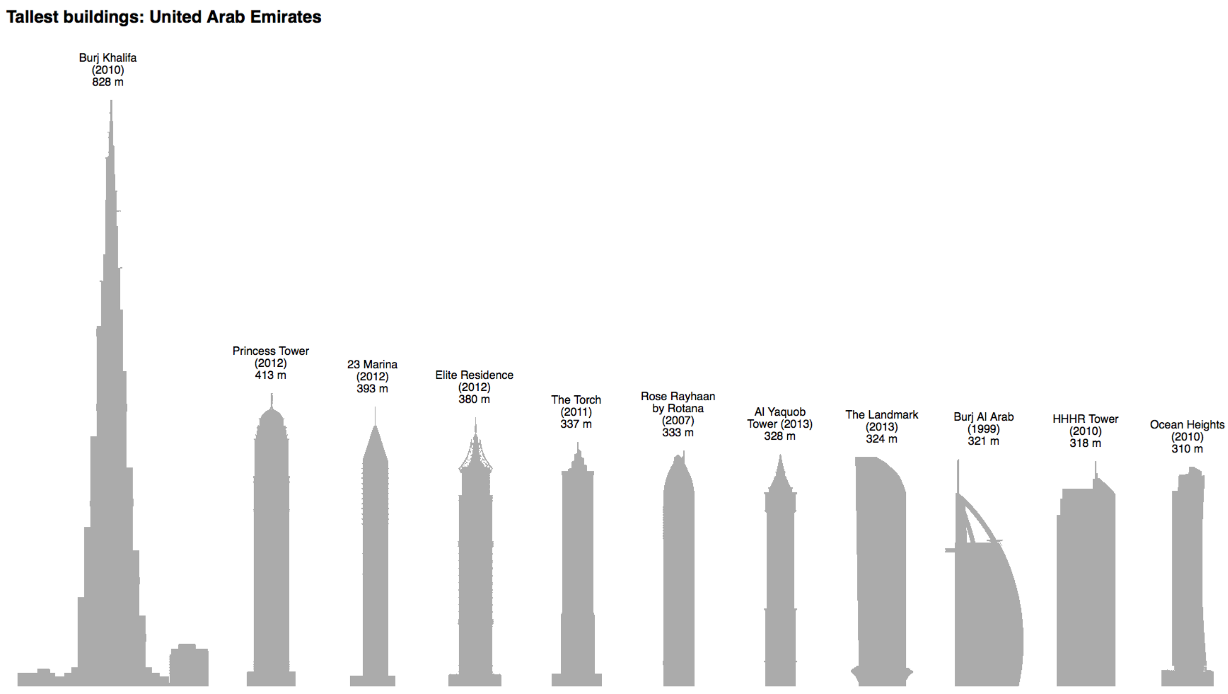
All structures are grouped by country, and the countries are sorted by the name of the country in the current locale; English in this example. The next step is to create a line up of all tall structures, tallest structures first. Note also that for formatting the height a localization function is used to present the height in meters according to the customs of the locale.

The final rendered image can be found in Figure 4.

Figure 3. Tallest Structure Code

```
<for-each select="doc('buildings.xml')/buildings/structure[status = 'completed']"
          group-by="location/country" sort="localize-country(location/country)">
  <h2>Tallest buildings: {{localize-country(current-grouping-key())}}</h3>
  <div class="structures-chart" id="{current-grouping-key()}">
    <for-each select="current-group()" sort="height/meters"
              sort-order="descending" sort-data-type="number">
      <div class="structure">
        <div class="structure-content">
          <div class="structure-label">
            <p class="structure-name">{{name}} ({{year}})</p>
            <p class="structure-height">{{format-length(height/meters, 'meter')}}</p>
          </div>
          <img src="{silhouette/@href}" alt="building silhouette"/>
        </div>
      </div>
    </for-each>
  </div>
</for-each>
```

Figure 4. Tallest Structures Rendered



5. Using Frameless in addition to NoSQL databases

Many organizations work hard to provide indiscriminate access to their websites, regardless of location, ethnicity, physical ability or economic status. In this regard, the coming years there is likely to be a shift towards "offline first" development, where a reliable internet connection is no longer assumed.

The offline first approach will prove to be very challenging because many kinds of complex computations that are usually conveniently performed by database servers, must be carried out entirely using just JavaScript.

In this area Frameless can be especially helpful bridging the gap between client and server capabilities, by providing the same technologies that are already in use on NoSQL servers such as MarkLogic and Zorba.

6. The road ahead

Within the next couple of months Frameless will be ready for us to be used for building a product we have been long looking forward to: a modular word processor. We will also be working with partners who use Frameless for their own application development, and improving the framework where needed.

For example: because not all data can be made available as XML, XPath queries in Frameless can also be used for JSON. Frameless is currently mostly optimized for XML so we feel that in the next year an improvement of the current JSON support is in order.

In addition to things that already were on the roadmap, like extending the documentation with more code samples, and Node.js support (allowing you to render the same HTML templates both in the browser and server-side) we have concrete plans to improve localization and performance of Frameless web applications.

Not only do we want to provide text translations inside templates, we also want to implement Unicode CLDR modules for formatting dates, numbers, et cetera. And of course we think users should be able to switch to another locale instantly.

Unfortunately, with every feature we add the download size of Frameless will increase. That's why we're working towards a module system, so every project can only include the functionality it needs, resulting in even smaller JavaScript downloads.

7. Summary

Reactive HTML templates excel at reducing development time, by offering a simple and intuitive syntax while offering a very powerful query language to interactively group, sort and filter data. Combined with very user-centric technologies such as full text search and localization, nothing should stand in the way of you making the spiffy next generation of Wikipedia or Google Docs. *Viva la revolución!*

Product Usage Schemas

Jorge Luis Williams

Rackspace Hosting

<jorge.williams@rackspace.com>

Abstract

In this case study we describe the process of collecting, validating, and aggregating usage information in a large public cloud for the purpose of billing. We also describe the Product Usage Schema a simple xml schema language used in-house to describe, version, and validate usage messages as they are emitted by various products across our public cloud.

Keywords: Usage Collection, Usage Validation, ATOM Syndication, XML Schema, WADL, XSLT, Cloud, Utility Computing

1. Background

The advent of cloud computing has created a shift in IT in which users may provision (or deprovision) computing infrastructure and software platform services on demand based on dynamic workloads. These on-demand computing resources may be hosted on premise (*private cloud*), off premise (*public cloud*), or may reside both internally and externally (*hybrid cloud*). Hybrid clouds allow for *cloud bursting*, the ability to scale work to a public cloud when a workload exceeds the capacity of a private cloud system.

Most cloud computing platforms make extensive use of concepts expressed by *Service Oriented Architecture* (SOA). In particular, they expose individual products as separate loosely coupled web services. Often these services are written using the *REST* architectural style as described by [6]. Here, IT resources such as virtual compute nodes, load balancers, databases, file storage volumes, LAMP stacks, and so forth are mapped to URIs accessible on the Internet (or via a private network). Operations on those resources are expressed via the uniform interface provided by the HTTP protocol. Thus, the provisioning, deprovisioning, and customization of resources can be achieved dynamically and with relative ease by simply performing HTTP requests on a resource URI. Likewise, the state of a resource may be monitored via HTTP's uniform interface — that is by performing a `GET` on that resource's URI. This accessible interface coupled with the ability to dynamically monitor and control resources are important keys to achieving cloud bursting.

Public cloud providers offer a utility computing model [5] to their customers in which the customer pays only for the actual use of the resources he or she consumes. This utility model requires the provider to implement metering of their services in order to track customer usage and produce monthly invoices. Resources are often sold at different pricing tiers which we refer to as *flavors*. Some flavors are higher performing or offer additional capabilities and are therefore more expensive than others. For example, virtual machines may be sold in a one gigabyte of RAM flavor or in a 16 gigabyte of RAM flavor. The 16 gigabyte flavor is significantly more expensive. A customer is allowed to change a resource's flavor at will — this is known as *resizing*. The resize operation may have the effect of upgrading or downgrading a resource and as a result a resource may be charged at different rates during its lifetime.

doi:10.14337/XMLLondon14.Williams01

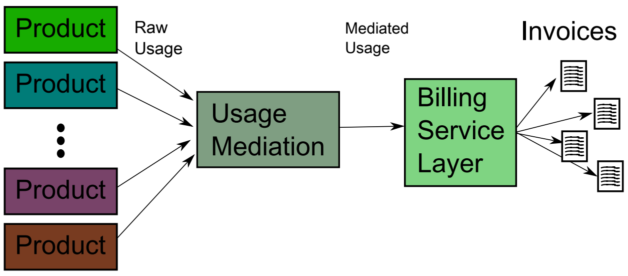
Cloud providers allow customers to provision resources at different geographical regions. The ability to control the relative location of a resource allows the customer to control network latency, build applications with a higher level of availability and fail over, and abide by governmental restrictions in the storage of sensitive data. It is important to note that the exact location of the data center in which a resource is provisioned is often obfuscated by a service provider because it is considered a security risk to expose it. At Rackspace, for example, we identify regions by using airport codes: DFW, LON, HKG. Also, note that a region may encompass multiple data centers. While customers may control the region in which a resource may be provision, the service provider controls the actual data center in which the resources lives.

The process of creating an invoice for a customer can be broken into a number of distinct parts. First, usage data must be collected from individual product services across all regions and aggregated along a number of dimensions such as:

- The owner of the resource (known as the *tenant*).
- The resource itself (the individual virtual compute node, load balancer, etc.).
- And the billable usage created by the resource — the usage type (CPU cycles, bandwidth, etc).

The aggregation process also involves producing daily summaries of these properties and enriching the data by adding additonal attributes such as a unique billable account number. We refer to the process of collecting, aggregating, summarizing, and enriching as *mediation*. As a result of the mediation process *raw usage* is converted into *mediated usage*. Finally, mediated usage is consumed by a number of billing services responsible for rating the usage for a particular billing cycle, applying promotions and discounts, and producing a final bill. Together these services form what we refer to as a *billing service layer*. An example billing pipeline is illustrated in Figure 1, "Billing Pipeline".

Figure 1. Billing Pipeline



2. Rationale and First Steps

Our initial billing pipeline required the usage mediation system to consume raw usage data directly from underlying product implementations. Products were in complete control of the format they used to produce this data. This solution caused a number of problems:

- Common cross-product attributes differed from one product to the next and the usage mediation team had to keep track of those differences. These differences could be very subtle, for example one product may emit times in central standard time (CST) another in GMT.
- Attributes described in raw usage data are likely to change as a product evolves. None-the-less, there was no process by which the usage mediation team could be notified of those changes.
- Because raw usage data was not required to adhere to any particular schema, it was difficult to catch data generation errors. Often subtle errors found their way to production systems — even after extensive testing.
- In cases where errors were identified in production, they were often identified in the later stages of the billing pipelines, at times after a customer has been billed. Remediation required the reprocessing of a large amounts of data and in the worst case required Rackspace to either compensate a customer or absorb a loss.

To address these issues we undertook a standardization effort in order to bring consistency, strict validation, and versioning of raw usage data into our billing pipeline.

2.1. Using AtomPub

Rather than having the usage mediation system query all products for raw usage, the new pipeline required products to emit usage to a centralized system using the Atom Publishing Protocol (AtomPub) [4]. We selected AtomPub for a number of reasons:

- We required a RESTful solution and the AtomPub protocol is considered an authoritative example of a RESTful protocol.
- The protocol and the underlying Atom format [1] are backed by a large number of implementations. On this front, Rackspace had already developed an implementation of an AtomPub sever, Atom Hopper [3], which we could easily leverage.
- The Atom format is extensible, so it would be trivial to extend the protocol to suite our needs.

- There is a standard AtomPub extension to support archiving [2]. This solution provides us with a good model for storing and providing access to long term archives of usage data for auditing purposes.

In the AtomPub model, usage data is packaged into discrete events in Atom Entries. These entries are collected in feeds where we reserved one feed for product. These feeds are mapped to a specific URI using a shared endpoint. For example, all usage data for our load balancer product is submitted to `/lbaas/events`, usage data for our database product is sent to `/dbaas/events` and so forth.

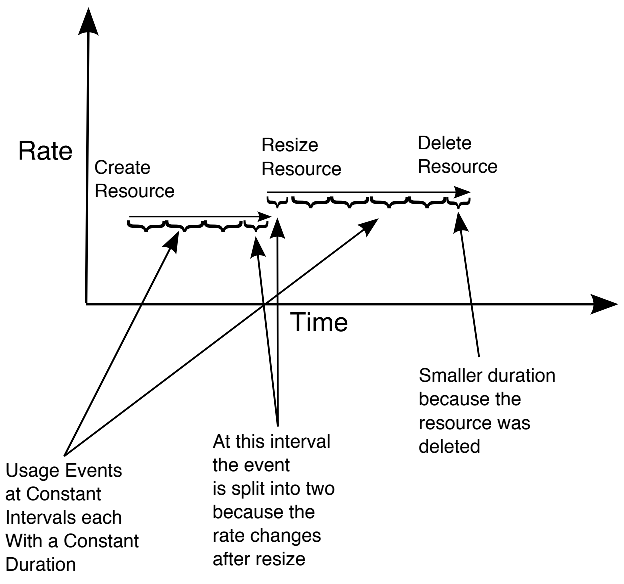
Note

Lbaas translates to Load Balancer As A Service and Dbaas translates to Database As A Service. We refer to some products by their internal product name, for example the compute service is known internally as Nova, so the atom feed is mapped to `/nova/events`.

In order to avoid network delays we deploy one instance of Atom Hopper per region. The regional product service is then responsible for submitting usage to the local instance. For example, usage for a load balancer in the London region is submitted to `http://feeds.lon.rackspace.com/lbaas/events` and usage for load balancer in the Dallas region is submitted to `http://feeds.dfw.rackspace.com/lbaas/events`.

Each atom feed accepts two distinct types of events: `USAGE` events and `USAGE_SNAPSHOT` events. `USAGE` events capture the utility of a resource over a time duration. The are usually emitted at constant intervals throughout the day but they may also be emitted after a resize operation or when a resource is deleted. This is illustrated in Figure 2, “`USAGE` Events Emitted for a Resource”. `USAGE_SNAPSHOT` events denote cases which do not correspond directly with the utility model such as one time charges and subscriptions. These events are emitted as needed on an ad hoc basis.

Figure 2. `USAGE` Events Emitted for a Resource


Rate
Create Resource
Resize Resource
Delete Resource
Time
Smaller duration because the resource was deleted
At this interval the event is split into two because the rate changes after resize
Usage Events at Constant Intervals each With a Constant Duration

2.2. The Usage Event Format

Products are required to submit usage events in a standard format. A standard raw usage event is made up of two elements: an `event` element contains attributes which are common across all products, and a `product` element which contains product specific attributes. A load balancer usage event is illustrated in Example 1, “Load Balancer Usage Event”.

Example 1. Load Balancer Usage Event

```
<?xml version="1.0" encoding="UTF-8"?>
<atom:entry xmlns="http://docs.rackspace.com/core/event"
          xmlns:atom="http://www.w3.org/2005/Atom"
          xmlns:lbaas="http://docs.rackspace.com/usage/lbaas">
  <atom:title type="text">LBAAS</atom:title>
  <atom:content type="application/xml">
    <event type="USAGE" version="1"
         tenantId="3737"
         resourceName="MyLoadBalancer"
         endTime="2012-06-15T10:19:52Z"
         startTime="2012-06-14T10:19:52Z"
         region="DFW" dataCenter="DFW1"
         id="b79cc3de-b399-3883-b555-61829bb7f966"
         resourceId="b79cc3de-b399-3883-b555-61829bbccd38">
      <lbaas:product serviceCode="CloudLoadBalancers"
                     resourceType="LOADBALANCER" version="1"
                     sslMode="MIXED" vipType="PUBLIC" numVips="44"
                     numPolls="10"
                     bandWidthOutSsl="345345346" bandWidthInSsl="364646770"
                     bandWidthOut="3460346" bandWidthIn="43456346"
                     avgConcurrentConnectionsSsl="4566.0"
                     avgConcurrentConnections="30000.0"
                     status="ACTIVE"/>
    </event>
  </atom:content>
</atom:entry>
```

Both the `event` and the `product` element have a set of required attributes:

Required `event` attributes

`id`
: A unique identifier for the event.

`type`
: The type of usage event. May be one of `USAGE` or `USAGE_SNAPSHOT`.

`version`
: A version number for the event format.

`tenantId`
: The owner of the resource.

`region` and `datacenter`
: The specific location of the resource.

`resourceId`
: A unique identifier for the resource.

`resourceName`
: The name of the resource as given by the tenant. This will be used in the invoice line item.

`startTime`, `endTime` or `eventTime`
: For a `USAGE` event `startTime` and `endTime` are required and represent the duration for the specified usage. For a `USAGE_SNAPSHOT` event `eventTime` is required.

Required `product` attributes

`serviceCode`
: A unique identifier for the product.

`resourceType`
: The type of resource that is being used.

`version`
: A version number for the product element format.

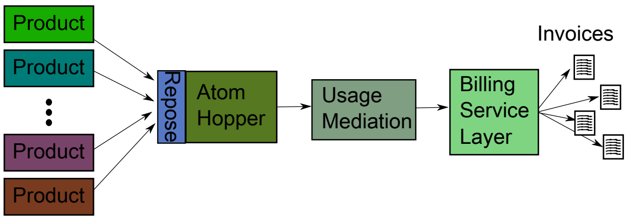
While an `event` element is allowed to contain only the required attributes above, the `product` element may contain additional product specific attributes. It is important to note that the `product` element is defined in a product specific namespace. In Example 1, “Load Balancer Usage Event”, the namespace is `http://docs.rackspace.com/usage/lbaas`. A product may define more than one type of `USAGE` event, each with a `product` element in a separate namespace. The `resourceType` and `serviceCode`, attributes are defined in the `product` element because this enables product specific validation. In other words, by defining these attributes in the `product` element the product can specify a finite set of possible attribute values as part of the product-specific element definition.

2.3. Validating Events

Validation of usage events is achieved by employing Repose [8]. Repose is a programmable HTTP proxy which sits in front of most of our REST services, it is capable of extending a service's capabilities by implementing common tasks such as authorization, rate limiting, transformation and validation of requests. We augmented our billing pipeline by deploying an instance of Repose in front of our Atom Pub server as illustrated in Figure 3, "New Billing Pipeline".

Repose can load a description of a REST service in WADL format [7] and reject requests which do not conform to the contract. We described the process that Repose employs for WADL validation in detail here: [10]. The WADL that we use to validate our usage events looks similar to the one in Example 2, "Usage Validation WADL for Atom Hopper".

Figure 3. New Billing Pipeline



Example 2. Usage Validation WADL for Atom Hopper

```
<?xml version="1.0"?>
<application xmlns="http://wadl.dev.java.net/2009/02" xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:atom="http://www.w3.org/2005/Atom">
  <grammars>
    <include href="core_xsd/entry.xsd"/>
  </grammars>
  <resources base="http://localhost/">
    <resource path="autoscale/events"  type="wadl/feed.wadl#AtomFeed wadl/feed.wadl#Unvalidated"/>
    <resource path="backup/events"     type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudBackup"/>
    <resource path="bigdata/events"    type="wadl/feed.wadl#AtomFeed wadl/product.wadl#BigData"/>
    <resource path="cbs/events"        type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudBlockStorage"/>
    <resource path="dbaas/events"      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudDatabase"/>
    <resource path="dns/events"        type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudDNS "/>
    <resource path="domain/events"     type="wadl/feed.wadl#AtomFeed wadl/product.wadl#DomainRegistration"/>
    <resource path="ebs/events"        type="wadl/feed.wadl#AtomFeed wadl/product.wadl#EBS"/>
    <resource path="emailapps/events"  type="wadl/feed.wadl#AtomFeed wadl/product.wadl#EmailApps"/>
    <resource path="files/events"      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudFiles"/>
    <resource path="glance/events"     type="wadl/feed.wadl#AtomFeed wadl/product.wadl#Glance"/>
    <resource path="lbaas/events"      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudLoadBalancers"/>
    <resource path="meta/events"       type="wadl/feed.wadl#AtomFeed wadl/product.wadl#AtomHopper"/>
    <resource path="monitoring/events" type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudMonitoring"/>
    <resource path="netdevice/events"  type="wadl/feed.wadl#AtomFeed wadl/product.wadl#NetDevice"/>
    <resource path="nova/events"
      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudServersOpenStack wadl/product.wadl#RHEL"/>
    <resource path="queues/events"     type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudQueues"/>
    <resource path="servers/events"
      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudServers wadl/product.wadl#RHEL"/>
    <resource path="sites/events"      type="wadl/feed.wadl#AtomFeed wadl/product.wadl#CloudSites"/>
    <resource path="ssl/events"        type="wadl/feed.wadl#AtomFeed wadl/product.wadl#Ssl"/>
    <resource path="support/events"    type="wadl/feed.wadl#AtomFeed wadl/product.wadl#Support"/>
  </resources>
</application>
```

Each Atom feed is represented by a resource element in the WADL. The element is associated with at least two resource types which are defined externally either in wadl/feed.wadl or in wadl/product.wadl. The resource type wadl/feed.wadl#AtomFeed contains common properties of a feed. For example, a feed accepts a GET operation and a feed can contain a list of entries which may be accessed independently. This resource type is used by all product feeds. The product WADL, on the other hand, defines product specific properties and validations. These product specific resource types prevent the cross posting of usage events. A usage event from the load balancer service, for example, is not acceptable in the DNS feed. Note that several feeds may accept the same product resource types. In the example above, both the OpenStack compute service (nova/events) and our legacy compute service (severs/events) support the resource type wadl/product.wadl#RHEL which contains rules for processing usage messages for RedHat Enterprise Linux (RHEL) licenses. Finally, there is a special resource type wadl/feed.wadl#Unvalidated, this type loosens product specific validations and is used when on-boarding a new product before product validations are set.

Note that the WADL makes reference to an XML schema document (XSD) [11] core_xsd/entry.xsd. This document contains the definition of the atom entries which contain usage events. We create a special complex type for the atom content element that specifically refers to our definition of a usage event element as illustrated in Example 3, "Atom Entry Usage Content Type". This allows the the schema processor to strictly validate the event element.

Example 3. Atom Entry Usage Content Type

```
<complexType name="UsageContent">
  <complexContent>
    <extension base="atom:BaseContent">
      <sequence>
        <choice>
          <element ref="event:event"/>
        </choice>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

In the definition of the event element itself, we allow for an element in a foreign namespace which must be strictly validated against a schema. Thus aside from defining the common cross-product attributes in Required event attributes, the definition of event contains the following:

```
<sequence>
  <any namespace="##other" processContents="strict"
       minOccurs="0" maxOccurs="unbounded"/>
</sequence>
```

This allows products to define their own product elements with an XSD. These elements must contain the attributes in Required product attributes, though we provide no explicit constrantraint on the XSD to enforce this requirement. Additonally, the product XSDs are expected to annotate custom product attributes with instructions that help guide the mediation process. For example, the definition of the avgConcurrentConnections attribute in Example 1, "Load Balancer Usage Event" is defined as:

```
<attribute name="avgConcurrentConnections"
           use="required"
           type="p:avgConcurrentConnections1Type">
  <annotation>
    <documentation>
      <html:p>
        The amount of conncurrent connections.
      </html:p>
    </documentation>
    <appinfo>
      <usage:attributes
        aggregateFunction="WEIGHTED_AVG"
        unitOfMeasure="COUNT"
        groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
```

Here, the `usage:attributes` annotation denotes that values from this attribute should be aggregated using a weighted average function, that the unit of measure for these attributes is simply a count and that attributes with the exact same value should not be grouped together. Likewise the `bandwidthIn` attribute is defined as:

```
<attribute name="bandWidthIn" use="required"
           type="p:bandWidthIn1Type">
  <annotation>
    <documentation>
      <html:p>
        The amount of bandwidth in, in bytes.
      </html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="SUM"
        unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
```

Here, the unit of measure is bytes and the values should be summed together during aggregation. The goal of these annotations is to enable the mediation process to dynamically adjust to product changes. Additionally, there is the potential to reduce errors by placing declarative control of the meditation process in the hands of product owners who best understand their product.

3. The Product Schema

Our initial implementation involved the manual creation of product specific XSDs given the rules defined in Section 2.3, "Validating Events". Additionally, the product specific resource types defined in `wadl/product.wadl` were manually maintained. This was a tedious and error prone process. Since product teams owned their product specific XSDs, it required them to have detailed knowledge of XML schema something many teams were not familiar with. What's more, the process of checking that a product schema conformed to the rules we defined in terms of required attributes and annotations was mostly a manual one. To help automate the process, we developed a simplified schema format to represent product specific attributes. We refer to documents in this format as *Product Schemas*. We developed an XProc [9] pipeline to generate product specific XSDs and the product WADL from a collection of these schemas.

3.1. The Product Schema Format

The product schema for the usage message in Example 1, "Load Balancer Usage Event" is illustrated in Example 4, "Load Balancer Usage Product Schema".

Example 4. Load Balancer Usage Product Schema

```
<productSchema xmlns="http://docs.rackspace.com/core/usage/schema"
               namespace="http://docs.rackspace.com/usage/lbaas"
               serviceCode="CloudLoadBalancers"
               version="1"
               resourceTypes="LOADBALANCER">
    <description>
        Lbaas load balancer usage fields.
    </description>
    <attribute name="avgConcurrentConnections" type="double" use="required"
               aggregateFunction="WEIGHTED_AVG"
               unitOfMeasure="COUNT" min="0" max="1000000">
        The amount of conncurrent connections.
    </attribute>
    <attribute name="avgConcurrentConnectionsSsl" type="double" use="required"
               aggregateFunction="WEIGHTED_AVG"
               unitOfMeasure="COUNT" min="0" max="1000000">
        The amount of conncurrent ssl connections.
    </attribute>
    <attribute name="bandWidthIn" type="unsignedLong" use="required"
               unitOfMeasure="B" aggregateFunction="SUM" min="0"
               max="10995116277760">
        The amount of bandwidth in, in bytes.
    </attribute>
    <attribute name="bandWidthOut" type="unsignedLong" use="required"
               unitOfMeasure="B" aggregateFunction="SUM" min="0"
               max="10995116277760">
        The amount of bandwidth out in bytes.
    </attribute>
    <attribute name="bandWidthInSsl" type="unsignedLong" use="required"
               unitOfMeasure="B" aggregateFunction="SUM" min="0"
               max="10995116277760">
        The amount of ssl bandwidth in, in bytes.
    </attribute>
    <attribute name="bandWidthOutSsl" type="unsignedLong" use="required"
               unitOfMeasure="B" aggregateFunction="SUM" min="0"
               max="10995116277760">
        The amount of ssl bandwidth out in bytes.
    </attribute>
    <attribute name="numPolls" type="int" use="required"
               unitOfMeasure="COUNT"
               min="0" max="288">
        The number of polls per load balancer.
    </attribute>
    <attribute name="numVips" type="int" use="required"
               unitOfMeasure="COUNT"
               min="0" max="100">
        The number of vips per load balancer.
    </attribute>
    <attribute name="vipType" type="string" use="required"
               allowedValues="PUBLIC SERVICENET">
        The vip type associated with the load balancer.
    </attribute>
    <attribute name="sslMode" type="string" use="required"
               allowedValues="ON OFF MIXED">
        The mode determining SSL status on the load balancer.
    </attribute>
    <attribute name="status" type="string" use="required"
               allowedValues="ACTIVE SUSPENDED">
        Is the load balancer currently active?
    </attribute>
    <xpathAssertion test="if (@status = 'SUSPENDED') then
                          ((@bandWidthIn = 0) and (@bandWidthOut = 0) and
                          (@bandWidthInSsl = 0) and (@bandWidthOutSsl = 0) and
                          (@avgConcurrentConnections = 0) and
                          (@avgConcurrentConnectionsSsl = 0)) else true()">
        If the status is SUSPENDED then bandWidthIn, bandWidthOut,
        bandWidthInSsl, bandWidthOutSsl, avgConcurrentConnections, and
        avgConcurrentConnectinsSsl should all be 0.
    </xpathAssertion>
    <xpathAssertion test="if (@sslMode = 'OFF') then
                          ((@bandWidthInSsl = 0) and (@bandWidthOutSsl = 0) and
                          (@avgConcurrentConnectionsSsl = 0)) else true()">
        If SslMode is OFF then bandWidthInSsl, bandWidthOutSsl, and
        avgConcurrentConnectionsSsl should all be 0.
    </xpathAssertion>
    <xpathAssertion test="if (@sslMode = 'ON') then
                          ((@bandWidthIn = 0) and (@bandWidthOut = 0) and
                          (@avgConcurrentConnections = 0)) else true()">
        If SslMode in ON then bandWidthIn, bandWidthOut, and
        avgConcurrentConnections should all be 0.
    </xpathAssertion>
</productSchema>
```

Here, the Required product attributes are required attributes in the schema, an error is emitted if they are not defined. Note that the `resourceTypes` attribute is a white-space separated list of resource types — in this case, only one resource type is defined: `LOADBALANCER`. The product specific namespace is simply defined with the `namespace` attribute and a general description of the usage event is in the `description` element.

3.2. Attributes

Attributes in the product schema have the following properties:

- The embedded text in the `attribute` element serves as documentation for the attribute and is required.
- Mediation annotations are simply defined as part of the element (`aggregateFunction`, `unitOfMeasure`). Once we introduced this feature, the mediation team stopped parsing XSD annotations and began consuming our product schemas directly instead.
- Some of the attributes correspond directly with those defined in XSD. In particular `name`, `type`, `use`.
- Enumerations are defined via the `allowedValues` attribute. Here, allowed values are specified as white-space separated list of values. Enumerations work with all basic types — not just strings.
- Ranges are defined directly via the `min` and `max` attributes.
- The `type` attribute contains a finite set of types, most inherited directly from XSD. Except that:
  - We extend the list of available types to include simple types that are common in our APIs — such as UUID.
  - The string type always includes a maximum character length of 255 characters as this is required by the back-end billing service layer.
  - Date types are required to use the GMT timezone.
  - List types are supported and denoted with a star(*). For example `int*`, `long*`. Since lists are white-space separated attributes `string*` is not supported but `name*` is. Note that it is possible to mix list types with ranges and enumerations.

3.3. Assertions

As Example 4, “Load Balancer Usage Product Schema” illustrates, XPath validation assertions are supported. Assertions may have a `scope` attribute which denotes the current node of the XPath expression. Allowed values for `scope` are simply `entry` (the root element of an Atom event) and `product` (the element containing product specific attributes) with `product` being the default. An example of an assertion with an `entry` scope is illustrated in Example 5, “Example non-product Assertion”.

Example 5. Example non-product Assertion

```
<xpathAssertion
  test="$event/@resourceId castable as xs:integer"
  scope="entry">
    The resourceId for a VIP should be an integer.
</xpathAssertion>
```

How an assertion is translated depends on the `scope` attribute. Assertions which only affect the `product` element are embedded directly in the generated XSD as XSD 1.1 [12] assertions. Example 6, “Generated ComplexType for Load Balancer usage message” illustrates the ComplexType in the generated XSD for Example 4, “Load Balancer Usage Product Schema”. Note that, unfortunately, we must generate both a `xerces:message` attribute an a `saxon:message` attribute to ensure that a correct error message is displayed, when an assertion fails, regardless of which XSD implementation we're using — we continually test against both Saxon and Xerces implementations though we are currently utilizing Saxon in production. This sort of requirement illustrates the benefit of generating an XSD rather than developing it by hand.

Example 6. Generated ComplexType for Load Balancer usage message

```
<complexType xmlns="http://www.w3.org/2001/XMLSchema"
             xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
             xmlns:html="http://www.w3.org/1999/xhtml"
             xmlns:usage="http://docs.rackspace.com/core/usage"
             xmlns:xerces="http://xerces.apache.org"
             xmlns:saxon="http://saxon.sf.net/"
             xmlns:p="http://docs.rackspace.com/usage/lbaas"
             name="CloudLoadBalancers1Type">
  <annotation>
    <documentation>
      <html:p>Cloud Load Balancer usage fields.</html:p>
    </documentation>
    <appinfo>
      <usage:core groupByResource="true" ranEnrichmentStrategy="CI_SERVICE" type="USAGE"/>
    </appinfo>
  </annotation>
  <complexContent>
    <extension base="p:BaseCloudLoadBalancersType">
      <attribute name="resourceType" use="required" type="p:ResourceTypes1"/>
      <attribute name="avgConcurrentConnections" use="required" type="p:avgConcurrentConnections1Type">
        <annotation>
          <documentation>
            <html:p>The amount of conncurrent connections.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" unitOfMeasure="COUNT" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="avgConcurrentConnectionsSsl" use="required" type="p:avgConcurrentConnectionsSsl1Type">
        <annotation>
          <documentation>
            <html:p>The amount of conncurrent ssl connections.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" unitOfMeasure="COUNT" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="avgConcurrentConnectionsSum" use="optional" type="xsd:double">
        <annotation>
          <documentation>
            <html:p>The sum amount of conncurrent connections for regular and SSL.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="WEIGHTED_AVG" unitOfMeasure="COUNT" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="bandWidthIn" use="required" type="p:bandWidthIn1Type">
        <annotation>
          <documentation>
            <html:p>The amount of bandwidth in, in bytes.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" unitOfMeasure="B" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
```

```
<attribute name="bandWidthInSsl" use="required" type="p:bandWidthInSsl1Type">
  <annotation>
    <documentation>
      <html:p>The amount of ssl bandwidth in, in bytes.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="NONE" unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="publicBandWidthInSum" use="optional" type="xsd:unsignedLong">
  <annotation>
    <documentation>
      <html:p>The sum amount of bandwidth in for regular and SSL, in bytes.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="SUM" unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="bandWidthOut" use="required" type="p:bandWidthOut1Type">
  <annotation>
    <documentation>
      <html:p>The amount of bandwidth out in bytes.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="SUM" unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="bandWidthOutSsl" use="required" type="p:bandWidthOutSsl1Type">
  <annotation>
    <documentation>
      <html:p>The amount of ssl bandwidth out in bytes.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="SUM" unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="publicBandWidthOutSum" use="optional" type="xsd:unsignedLong">
  <annotation>
    <documentation>
      <html:p>The sum amount of bandwidth out for regular and SSL, in bytes.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="SUM" unitOfMeasure="B" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="numPolls" use="required" type="p:numPolls1Type">
  <annotation>
    <documentation>
      <html:p>The number of polls per load balancer.</html:p>
    </documentation>
    <appinfo>
      <usage:attributes aggregateFunction="NONE" unitOfMeasure="COUNT" groupBy="false"/>
    </appinfo>
  </annotation>
</attribute>
<attribute name="numVips" use="required" type="p:numVips1Type">
  <annotation>
```

```
          <documentation>
            <html:p>The number of vips per load balancer.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" unitOfMeasure="COUNT" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="vipType" use="required" type="p:vipType1Enum">
        <annotation>
          <documentation>
            <html:p>The vip type associated with the load balancer.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="sslMode" use="required" type="p:sslMode1Enum">
        <annotation>
          <documentation>
            <html:p>The mode determining SSL status on the load balancer.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="hasSSLConnection" use="optional" type="xsd:boolean">
        <annotation>
          <documentation>
            <html:p>An attribute determining whether or not the Cloud Load Balancer
                    used an SSL connection. Used for billing purposes.</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" groupBy="true"/>
          </appinfo>
        </annotation>
      </attribute>
      <attribute name="status" use="required" type="p:status1Enum">
        <annotation>
          <documentation>
            <html:p>Is the load balancer currently active?</html:p>
          </documentation>
          <appinfo>
            <usage:attributes aggregateFunction="NONE" groupBy="false"/>
          </appinfo>
        </annotation>
      </attribute>
      <assert vc:minVersion="1.1" test="if (@status = 'SUSPENDED') then
                                  ((@bandWidthIn = 0) and (@bandWidthOut = 0) and
                                  (@bandWidthInSsl = 0) and (@bandWidthOutSsl = 0) and
                                  (@avgConcurrentConnections = 0) and
                                  (@avgConcurrentConnectionsSsl = 0)) else true()"
              xerces:message="If the status is SUSPENDED then bandWidthIn, bandWidthOut, bandWidthInSsl,
                        bandWidthOutSsl, avgConcurrentConnections, and avgConcurrentConnectinsSsl
                        should all be 0."
              saxon:message="If the status is SUSPENDED then bandWidthIn, bandWidthOut, bandWidthInSsl,
                        bandWidthOutSsl, avgConcurrentConnections, and avgConcurrentConnectinsSsl
                        should all be 0.">
        <annotation>
          <documentation>
            <html:p>Assertion: If the status is SUSPENDED then bandWidthIn, bandWidthOut, bandWidthInSsl,
                    bandWidthOutSsl, avgConcurrentConnections, and
                    avgConcurrentConnectinsSsl should all be 0.</html:p>
          </documentation>
        </annotation>
      </assert>
      <assert vc:minVersion="1.1" test="if (@sslMode = 'OFF') then ((@bandWidthInSsl = 0) and
                                  (@bandWidthOutSsl = 0) and (@avgConcurrentConnectionsSsl = 0))
                                  else true()"
              xerces:message="If SslMode is OFF then bandWidthInSsl, bandWidthOutSsl,
                        and avgConcurrentConnectionsSsl should all be 0."
              saxon:message="If SslMode is OFF then bandWidthInSsl, bandWidthOutSsl,
                       and avgConcurrentConnectionsSsl should all be 0.">
        <annotation>
          <documentation>
            <html:p>Assertion: If SslMode is OFF then bandWidthInSsl, bandWidthOutSsl,
                    and avgConcurrentConnectionsSsl should all be 0.</html:p>
          </documentation>
        </annotation>
      </assert>
      <assert vc:minVersion="1.1" test="if (@sslMode = 'ON') then ((@bandWidthIn = 0)
                                  and (@bandWidthOut = 0) and
                                  (@avgConcurrentConnections = 0)) else true()"
              xerces:message="If SslMode in ON then bandWidthIn, bandWidthOut,
                              and avgConcurrentConnections should all be 0."
              saxon:message="If SslMode in ON then bandWidthIn, bandWidthOut,
                           and avgConcurrentConnections should all be 0.">
        <annotation>
          <documentation>
            <html:p>Assertion: If SslMode in ON then bandWidthIn, bandWidthOut,
                    and avgConcurrentConnections should all be 0.</html:p>
          </documentation>
        </annotation>
      </assert>
    </extension>
   </complexContent>
  </complexType>
```

Assertions with `scope` of `entry` which involve reaching into different parts of the message, cannot be generated directly into an XSD because the XSD specification dictates that elements containing an assertion are considered root elements. Because of this, assertions may not reach out to parent elements (although the Xerces XSD implementation currently allows this). These assertions are instead converted into an XSLT transformation [13] and embedded as part of the `product.wadl` (since Repose supports an extension to WADL which executes an XSLT before validating the message). These XSLT transforms perform the identity transformation, but fail if the XPath assertion is not satisfied. The translated assertion for Example 5, "Example non-product Assertion" is illustrated in Example 7, "Generated XSLT fragment for Load Balancer usage message assertion with entry scope".

Example 7. Generated XSLT fragment for Load Balancer usage message assertion with entry scope

```
<xsl:when
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:p="http://docs.rackspace.com/event/lbaas/lb"
  test="$event/p:product">
  <xsl:variable name="product"
                select="$event/p:product"/>
  <xsl:choose>
    <xsl:when test="$product[@version = '1']">
      <xsl:choose>
        <xsl:when test="$event/
              @resourceId castable as xs:integer"/>
        <xsl:otherwise>
          <xsl:message terminate="yes">
            The resourceId for a load balancer
            should be an integer.
          </xsl:message>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:when>
  </xsl:choose>
</xsl:when>
```

Note that the technique of translating a series of assertions into an XSLT for execution is similar to the technique used by Schematron [14]. The reason for generating an XSL rather than utilizing Schematron directly, in this case, is that validations need to occur at a very frequent interval (thousands of times a second) and the approach in Example 7, "Generated XSLT fragment for Load Balancer usage message assertion with entry scope" is much lighter weight. We do however, utilize Schematron to help validate WADLs when they are initially loaded by the Repose validator.

Also note that when the scope of the assertion is `entry` the variables `$entry`, `$event`, and `$product` are defined for convenience and the product namespace is always mapped to the prefix `p`. The prefixes `atom`, `event` and `xs` are also defined.

3.4. Versioning

It is possible to define multiple product schemas with the same `serviceCode` and `namespace` but with a different version number. In this case, our transformation creates a single XSD which supports multiple versions of a message by utilizing the `alternative` feature of XSD 1.1. The generated product element for our bigdata events illustrate this in Example 8, "Alternatives used for versioning".

Example 8. Alternatives used for versioning

```
<element name="product" vc:minVersion="1.1"
        type="p:BaseBigDataType">
  <alternative test="(@version eq '1')"
               type="p:BigData1Type"/>
  <alternative test="(@version eq '2')"
               type="p:BigData2Type"/>
</element>
```

The versioning feature allows multiple versions of a usage schema to co-exist in the feed. Our versioning strategy requires that versions remain backwards compatible as much as possible. The `version` attribute in a usage message is used to for validation purposes — it identifies which validation rules should apply. It does not imply that the usage messages are entirely incompatible from a client's perspective.

4. Future Work

- Our solution generates XSD 1.1 schemas, but some of our customers would like XSD 1.0 version because their XML binding tools such as JAXB [15] demand it. While the schemas that we generate take advantage of the conditional inclusion properties of XSD we still run into high levels of incompatibility. We plan on extending our system to generate (non-normative) XSD 1.0 schemas which are more useful.
- The process of validating usage events introduces little run-time overhead. On a production grade system, validation of a usage event takes about 2 milliseconds on average. This overhead has remained fairly stable even as we've introduced many different types of usage events. The initialization process of loading the generated WADL and its related resources (XSDs and XSLts) has been steadily increasing, however, and now averages about 5 minutes. These long load times mean that introducing new nodes or making making frequent schema changes, at run-time, incur a high cost and have the potential to affect our SLAs. We plan on investigating ways of decreasing these high initialization times — possibly by converting WADLs to an internal representation in an off-line preprocessing step.
- Since our product usage schema decouples us directly from XSD, we are exploring the possibility of supporting alternate formats for our usage messages including JSON. We would like to generate JSON schemas in the same way that we generate XML schemas.
- We are also exploring developing GUIs and wizards to aid products in the generation and maintenance of their product schemas.

5. Conclusion

In this case study, we examined the process of collecting, validating, and aggregating usage events in a large public cloud. We leveraged an XML based solution to transform our existing billing pipeline from one in which usage data was defined in irregular and inconsistent ways to one in which strict standards were followed and a declarative language was used to specify validation, versioning, and mediation rules. The new usage event model has allowed us to catch errors early which has significantly reduced the number and impact of errors on our customers. What's more, we've been able to achieve this strict level of validation with little to no negative impact on the overall performance of our system.

Our simple product schema format has allowed us to leverage the power of XSD and WADL without having to get lost in the idiosyncrasies of these languages. By doing so, we were able to design product specific usage data in a highly consistent manner and to communicate requirements to the usage mediation team accurately.

Bibliography

[1] *The Atom Syndication Format*. M. Nottingham and R. Sayre. https://tools.ietf.org/html/rfc5023

[2] *Feed Paging and Archiving*. M. Nottingham. https://tools.ietf.org/html/rfc5005

[3] *Atom Hopper*. http://atomhopper.org/

[4] *The Atom Publishing Protocol*. J. Gregorio and B. de Hóra. https://tools.ietf.org/html/rfc5023

[5] *IBM Systems Journal*. 43. 1. 2004. “Utility Computing”. J. Ritsko and B. Birman. http://www.research.ibm.com/journal/sj43-1.html

[6] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[7] M. Hadley. *Web Application Description Language*. http://www.w3.org/Submission/wadl/

[8] *Open Repose*. http://www.openrepose.org/

[9] N. Walsh, A. Milowski, and H. Thompson. *XProc: An XML Pipeline Language*. http://www.w3.org/TR/xproc/

[10] *Proceedings of Balisage: The Markup Conference 2012*. Balisage Series on Markup Technologies. 2012. 8. August 7 - 10, 2012. “Using XProc, XSLT 2.0, and XSD 1.1 to validate RESTful services.”. J. Williams and D. Cramer. doi:10.4242/BalisageVol8.Williams01

[11] *XML Schema*. http://www.w3.org/XML/Schema

[12] *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. D. Peterson, S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. Thompson. http://www.w3.org/TR/xmlschema11-2/

[13] *The Extensible Stylesheet Language Family (XSL)*. http://www.w3.org/Style/XSL/

[14] *Schematron*. http://www.schematron.com

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

An XML-based Approach for Data Preprocessing of Multi-Label Classification Problems

Eduardo Corrêa Gonçalves

Universidade Federal Fluminense (UFF)

<egoncalves@ic.uff.br>

Vanessa Braganholo

Universidade Federal Fluminense (UFF)

<vanessa@ic.uff.br>

Abstract

Most of the data mining tools are only able to work with data structured either in relational tables or in text files with one record per line. However, both kinds of data representation are not well-suited to certain data mining tasks. One example of such task is multi-label classification, where the goal is to predict the states of a multi-valued target attribute. This paper discusses the use of XML as an alternative to represent datasets for multi-label classification processes, since this language offers flexible means of structuring complex information, thus potentially facilitating the major steps involved in data preprocessing. In order to discuss from a practical point of view, we describe the steps of an experience involving the preprocessing of a real text dataset.

Keywords: Data Preprocessing, Text Categorization, Multi-Label Classification

1. Introduction

Multi-label classification (MLC) can be defined as the task of automatically assigning an object into multiple categories based on its characteristics [1]. One of the most common applications is text categorization, where the goal is to associate documents to various subjects. For example: a trained multi-label classifier could process the text summary of the movie "The King's Speech" and determine its genres as "Biography", "Drama", and "History". Besides text categorization, other important applications for MLC include functional genomics (determining the functions of genes and proteins) and the semantic categorization of images, video and music. The MLC problem is more challenging than the traditional single-label classification (SLC) in which objects can be associated with only a single target class. Typically, MLC applications have to deal with a huge number of possible label combinations. Considering a problem involving *q* labels, the size of the output space in MLC is *2^q* whereas it is just *q* in SLC. Another important issue concerns the existence of correlations between labels. For instance, a movie is unlikely to be simultaneously labelled as "Romance" and "Horror", since these two genres have a strong negative correlation. Thus, algorithms capable of modelling label correlations tend to be more accurate. Actually, recent research in MLC have primarily concentrated efforts on the development of scalable techniques for modelling label dependencies [2], [3], [4].

However, an important characteristic that differentiates MLC and SLC has been often neglected in literature. It corresponds to the fact that real-world multi-label datasets (e.g. text data, multimedia data, etc.) are usually much larger and more complex in structure than single-label datasets. In spite of this situation, current multi-label platforms [5], [6] have adopted the traditional ARFF flat file format ("one record per line") to represent target datasets. This leads to two main problems: (i) the format is unnatural for the representation of multi-label data; (ii) the flat file format makes data preprocessing activities considerably more cumbersome, since it is not suitable for querying and transformation.

In other to tackle these problems, this paper proposes an XML-based approach for data representation and preprocessing in multi-label classification. The main goal is to discuss the advantages offered by this approach to users of data mining tools, focusing on the text

doi:10.14337/XMLLondon14.Goncalves01

categorization problem. The rest of this paper is organized as follows. Section 2 presents a comparison between the ARFF and the XML formats, highlighting the advantages associated with the use of the latter format. Section 3 describes an experiment that applied the XML approach for data preprocessing of a real-world dataset with movie information. Section 4 summarizes this work and points to future research directions.

2. Dataset Formats: ARFF versus XML

Mulan [6] and MEKA [5] are the two most used platforms for multi-label classification in research projects. Both work on the top of the well-known Weka API [7] and adopt the Weka's standard ARFF format for dataset representation. This format is simple, intuitive, having become popular in the data mining field. Even so, it is not suitable for representing multi-label datasets. To support this claim, consider the following example. Suppose we want to perform the multi-label classification of a movie database. The goal is to classify the genres of movies in function of the movie summaries. In this case, both Mulan and MEKA would require an ARFF dataset structured similarly to the example below.

```
@relation Movies
@attribute a {0, 1}
@attribute abandon {0, 1}
@attribute about {0, 1}
...
@attribute zoology {0, 1}
@attribute genre_drama{0, 1}
@attribute genre_mystery{0, 1}
...
@attribute genre_romance{0, 1}
@data
0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,1,...
1,0,0,0,1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,...
...
```

Observe that the ARFF format requires the declaration of every distinct word as a different attribute in the header. The data itself is structured in the bag of words (BOW) format. Each line contains information of a different movie. The 1 and 0 values represent, respectively, the presence and the absence of the correspondent word in the summary. Although the Weka API offers filters for the automatic conversion of free text into BOW, this conversion represents just a small part of the data preprocessing scenario. Typically, users of data mining tools need to explore and transform data before executing the classification algorithm. The BOW format is rather inadequate for querying and transformation though.

XML is a language for specifying semi or completely structured data adopted as a standard by many industries [8]. In the below code we show the "movies dataset" now structured in XML format. We can immediately realize that XML allows for the representation of movie information in a very natural way: each movie is simply delimited by the tags <movie> and </movie>. More interestingly, the definition of multi-valued data, i.e., movies with multiple summaries (<plot> tag) and genres (<class> tag), can be done in a quite straightforward fashion (differently from the ARFF format). However, the major advantage is that XML is able to potentially enhance data preprocessing procedures. Through an XML environment, users of a data mining tool can visualize, query, explore, and transform data in a simpler and faster way, by the means of the standard languages XSLT [9] and XQuery [10] or using the SAX API [11]. In the next section, we discuss the advantages of the XML-based data preprocessing approach from a practical point of view, describing an experiment performed over a real-world text dataset.

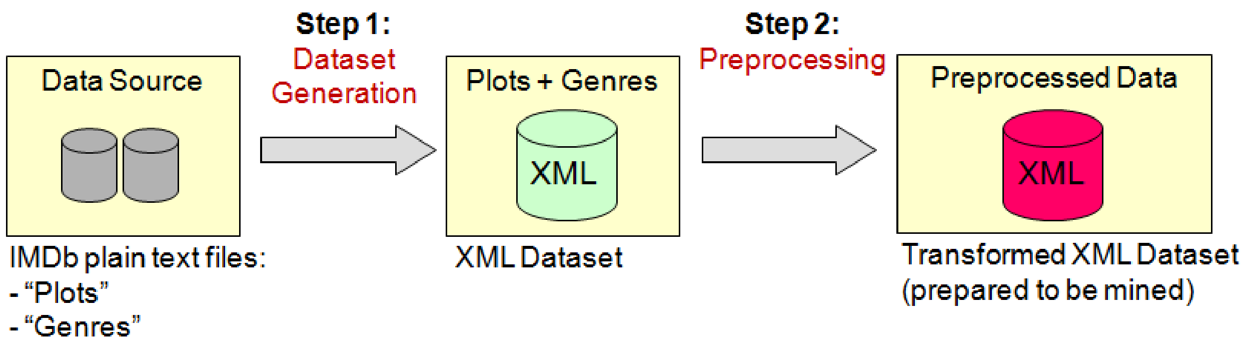
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<imdb>
  <movie id="500">
     <title>127 Hours</title>
     <plot>127 Hours is the true story of ...
     <plot>On April 2003, the engineer, climber ...
     <class>Adventure</class>
     <class>Biography</class>
     <class>Drama</class>
     <class>Thriller</class>
  </movie>
  <movie id="501">
     <title>The Bridges of Madison County</title>
     <plot>Photographer Robert Kincaid wanders ...
     <class>Drama</class>
     <class>Romance</class>
  </movie>
  ...
</imdb>
```

3. Experiment

The IMDB dataset [12] is real-world dataset that keeps information about movies. Data is originally supplied in several plain text files [1] which are weekly updated with new information. Each file stores different information regarding the movies. The experiment described in this section involved the files "plot.list" (movie summaries) and "genres.list" (movie genres). The goal of our evaluation was to assess the advantages of using XML representation in the data preprocessing step (step 2 in Figure Figure 1, "Proposed steps for the construction of the multi-label classifier for IMDb data"), supposing that a multi-label classifier to associate movies to genres would be constructed.

[1] http://www.imdb.com/interfaces

Figure 1. Proposed steps for the construction of the multi-label classifier for IMDb data



The first step of the experiment consisted in the development of a simple Java program called "imdb2XML" to generate the XML dataset from the plot and genres plain files. The structure of this dataset is similar to the one shown in the movies XML file presented in the last section. The final XML dataset was composed by 153.499 movies, which corresponds to the number of movies that are stored in the plot file and also in the genres file. The second step represents the main goal of this paper: the defence of an XML-based data preprocessing environment for multi-label classification problems. During the experiment, at this step, the XQuery language and the SAX API were used to querying, exploring and transforming the XML IMDb dataset. Below, we list some of the several preprocessing procedures that were performed. Observe that most of these procedures cannot be directly performed over ARFF datasets.

- Construction of a frequency table of movie genres. Among the 28 possible genres, some of the most frequent are "Drama" (59,177), "Comedy" (38,377), and "Documentary" (27,590).
- Generation of a word frequency table. This table was queried in several different ways. The query results were used to collect information about data and to guide data transformation operations over the XML dataset. Some examples:
  - About half of the words occurs only once in the database (e.g. "agnosticism", "polyvision"). These words were removed from the dataset.
  - Several misspelled words and typos (e.g. "caracters", "theforce").
  - There is a large number of proper names, e.g. "Robert" (3,053), "Rosemary" (229).
  - Construction of a cross-tabulation table of words and genres. This table has helped us to identify the words that are most correlated to certain genres (ex: word "Broadway" and genre "Musical").
- Calculation of the correlation coefficient between all pairs of labels (e.g.: "Drama" x "Action", "Drama" x "Horror", etc.) to identify labels that are positively and negatively correlated.

4. Conclusions

Multi-label classification is a challenging problem in the field of data mining, which requires considerable efforts in data preprocessing. In this paper we discussed the use of an XML-based approach as a feasible and adequate alternative for data preprocessing of multi-label datasets. The advantages of XML could be evidenced in distinct two forms: first, through a comparison between the formats XML and ARFF; second, through an experiment involving a real text dataset. In this experiment, the XQuery language and the SAX API were employed to perform data exploration and transformation. As future work we intend to conduct new evaluations on different multi-label datasets aiming at developing a general XML-based framework for data preprocessing.

Bibliography

[1] *A tutorial on multi-label classification techniques*. Andre de Carvalho and Alex Freitas. Foundations of Computational Intelligence, 5, 2009. ISBN: 978-3-642-01535-9. doi:10.1007/978-3-642-01536-6_8

[2] *A genetic algorithm for optimizing the label ordering in multi-label classifier chains*. Eduardo Corrêa Gonçalves, Alexandre Plastino, and Alex Freitas. ICTAI 2013. ISBN: 978-1-4799-2971-9. doi:10.1109/ICTAI.2013.76

[3] *Classifier chains for multi-label classification*. Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Machine Learning 85(3). ISSN: 0885-6125. doi:10.1007/s10994-011-5256-5

[4] *Incorporating label dependency into the binary relevance framework for multi-label classification*. Everton Alvares-Cherman, Jean Metz, and Maria Carolina Monard. Expert Systems with Applications 39(2). ISSN: 0957-4174. doi:10.1016/j.eswa.2011.06.056

[5] *MEKA - A multilabel/multitarget extension to WEKA*. SourceForge.net. http://meka.sourceforge.net/

[6] *Mulan: A Java library for multi-label learning*. SourceForge.net. http://mulan.sourceforge.net/

[7] *Weka 3: data mining software in Java*. University of Waikato. http://www.cs.waikato.ac.nz/ml/weka/

[8] *XML: some papers in a haystack*. Mirella Moro, Vanessa Braganholo, Carina Dorneles, Denio Duarte, Renata Galante, and Ronaldo Mello. ACM SIGMOD Record 38(2). ISSN: 0163-5808. doi:10.1145/1815918.1815924

[9] *XSL Transformations (XSLT) Version 1.0*. W3C. http://www.w3.org/TR/1999/REC-xslt-19991116

[10] *XQuery 1.0: An XML query language (second edition)*. W3C. http://www.w3.org/TR/xquery/

[11] *SAX 2.0.1: Simple API for XML*. David Megginson and David Brownell. SAX Project. http://www.saxproject.org/

[12] *IMDb – The Internet movie database*. IMDb.com, Inc.. http://www.imdb.com

Using Abstract Content Model and Wikis to link Semantic Web, XML, HTML, JSON and CSV

Using Semantic Media Wiki as a mechanism for storing format neutral content model

Lech Rzedzicki

<lech@kode1100.com>

1. Introduction

2013 has been hyped as the year of Big Data [1], 2014 is still about projects dealing with deluge of data and this trend is going to continue as organisations produce and retain exponentially growing amounts of data, outpacing their capability to utilise the data and gain insight from it.

One method of dealing with the data flood is modeling the data - applying rules to ensure it is consistent and predictable where possible, and flexible everywhere else, providing definitions, examples, alternatives and connecting related structures.

On one hand of the modeling spectrum is the traditional relational data modeling with conceptual, logical and physical models and levels of normalization. Such a strict approach is definitely working well in some environments, but not in publishing where requirements are in constant flux and are rarely well defined.

On the other hand of the spectrum is the 'NOSQL'[1] movement where data is literally dumped to storage as is and any data validation and modelling is kept in the application layer therefore needs software developers to maintain. At the moment NOSQL developers are a scarce minority amongst established publishers and a rare and expensive resource in general.

To balance these needs and problems, at Kode1100 Ltd[2] we have designed and developed a modeling system, which to a large extent is resilient to changes in developer fashion and taste and can be maintained by technically savvy and otherwise intelligent folks who do not have to be full time programmers.

2. Background

To understand better the problems Abstract Content Model is trying to solve we need to learn a little bit more about developer preferences and publishing industry.

I have found XML and XML validation languages like W3C (XSD) Schema, RelaxNG and Schematron to be excellent, flexible and powerful tools to express the rules about the data.

The fact that these technologies are international standards and have great community support means that it is easier to convince large organisations to invest time and money in such technology.

However, the developer world is much bigger and more fragmented that it was when standards like W3C Schema were formulated.

It is no longer reasonable to expect that everyone will go through the learning curve and adopt XML tools and techniques in their ecosystem.

Developers prefer to use software tools, libraries and techniques that they are familiar with, the ones they use anyway, everyday and they will apply the same principles to encoding data. Web Developers use JSON. Excel power users prefer CSV and .NET macros. Attendees of XML London hopefully still prefer XML.

Building a system that would cater to the needs of the above, existing and future groups was one of the core goals.

I will use a concrete use case to illustrate some of the concepts and techniques, but the same principles can be potentially applied to storing and interacting with a model of any structured or semi-structured data, especially if it is representable as XML.

[1] NoSQL - In Wikipedia. http://en.wikipedia.org/wiki/NoSQL
[2] kode1100.com - http://www.kode1100.com

doi:10.14337/XMLLondon14.Rzedzicki01

3. Technology Choices

Being an XML consultant I had a natural bias to use XML Technologies to develop the model, but as I explained in the background section one of the main goals for the Abstract Content Model is the ability to serve the needs of a variety of audiences.

My years of experience have taught that me that an open, widely adopted standard is the only sane choice for a long running project and fortunately the clients tend to agree.

We decided that RDF is the right format for the canonical version of the model: it is flexible enough to represent all the concepts but also an open standard that is adopted widely enough, but we needed input and output that is easier to work with than raw RDF-XML. The Semantic Web stack of technologies: RDF, SPARQL, RDFS, OWL etc are excellent tools for modeling and queries but have a pretty steep learning curve and face the problem of not being adopted widely enough. Some of the contributors to content model were expected not to have a background in software programming so we needed a layer of input and output that was easier to work with.

Therefore we also decided to use wiki software as a way to present and even edit the content model in a user friendly fashion as well as to document how the system works. Our initial user testing confirmed that editing wiki pages, especially with predefined forms was within capability of the user base.

We did a review of the available software and given that ability to export to RDF was a requirement, we had one obvious choice: Semantic Media Wiki.

MediaWiki is the open source software behind Wikipedia. It is therefore maintained by Wikimedia foundation and a wide community of open-source developers.

Semantic Media Wiki is a set of Extensions to Media Wiki to give it specific semantic capabilities to describe linked data- forms, infoboxes, RDF Export etc. It has a smaller but likewise responsive community. Semantic MediaWiki still has many unresolved bugs, very often around parsing strings, escaping characters and so on, but because it is written in PHP and open-source, a lot of developers can understand the code and contribute to improving the software. At least hypothetically.

With that setup (and a lot of customisation) we were able to export the whole wiki as RDF. It is certainly possible to just download the whole dump and parse that RDF file, but it is definitely not a scalable solution. In my previous project at OHIM, the size of the RDF dump exceeded 300MB in size.

We have therefore set up a Triple Store and a SPARQL endpoint. We first tried Jena and have found no fault with it- we haven't done any throughout testing of SPARQL endpoints side by side.

To compliment the setup and keep the stack fully Open Source, we are hosting it on Linux virtual machines (currently running Ubuntu 14) and using Apache HTTP server for complimentary functions.

The virtual machines are hosted at DigitalOcean and Rackspace, but because the deployment process is automated, it can be moved to another cloud provider such as Amazon, if they become cheaper or more cost effective.

4. Technical Description

The core concept of the Abstract Content Model is to abstract the canonical form of the content model, so that it is separated from any implementation. This allows the content to be neutral to the technology used in a actual production system that uses the model.

The content model is defined as an abstract graph of connected concepts and definitions which in turn have properties that can be abstract or more specific to a representation.

This maps really well to an RDF graph, but by adhering to conventions that graph can be simplified to a tree and then the whole model can then also be represented using hierarchical formats such as JSON or XML.

The implementation is also pretty simple (at least to an XML audience): the canonical form of the model is stored as RDF-XML and there are conversion pipelines to convert to the preferred representation format such as XML, JSON or CSV.

Because we have used Semantic MediaWiki as the software to interface with the content model and because XML is, at least for now, the most used input and output format, we have decided to optimise the content model for that.

Since XSD representation of the content model was a big requirement anyway and used extensively in the initial modelling, we decided to constrain Semantic MediaWiki content model to effectively mirror an XSD design pattern. I strongly recommend reading a bit more or refreshing your memory on XSD Design Patterns.

Initially we agreed on Venetian blinds pattern. It mapped really well to Object-like representations like JSON, it could potentially lead to being able to generate Object oriented code classes straight from the content model and it allowed to reuse the types. But in the end it was an abstraction on top of Semantic MediaWiki abstraction of a content model, which was to complicated and it also meant that the resulting RDF representation and therefore also SPARQL queries were unnecessarily complicated.

Our next and current approach is Garden of Eden and it is really simple. Every concept in the abstract content model maps to single page on the Semantic MediaWiki (and therefore it has it's own URI which is really useful for RDF) and is represented as a single element in XML Schema and XML instances.

The choice of Schema Design Pattern is merely a convention and was based on non-wiki related requirements, it may be more suitable for another similar project to adopt a different pattern to suit the requirements and such is the case with OHIM TM-XML and DS-XML which use Venetian Blinds patterns.

Semantic MediaWiki has implemented the concept of namespaces which is really useful to separate content model pages that are processed programmatically from templates, forms, auxiliary pages and just plain wiki pages that just serve as navigation or documentation. These namespaces, at least for now map 1:1 to namespaces in XML and RDF representations.

Another layer of separation is the Semantic MediaWiki concept of categories. All pages that map to XML Elements are in a Category:Elements. This makes Semantic MediaWiki Ask and SPARQL queries really easy and again makes RDF, XSD and JSON clean and easy to query or parse.

In addition to that setup we have developed a set of XProc pipelines to output to some popular and useful formats. The ones we developed initially were XML Sample, XSD, JSON, raw HTML, and HTML+RDFa.

The pipelines are written as a combination of SPARQL queries to grab a relevant subset of RDF (in RDF-XML representation) and then an XSL stylesheet to convert from RDF-XML to target format.

If the existing formats do not match the requirements, there is also a possibility for a system or a developer to work directly with the RDF.

For that we have set up automation to export RDF and load it to a triple store and expose a SPARQL endpoint. As discussed in the technology choices section, the triple store is Jena, which in addition to raw SPARQL endpoint offers a web form queries and, optionally a Linked Data API layer, to automatically offer RDF conversion to JSON, CSV and XML.

5. Content Modeling and everyday use

After installing and configuring the software, developing the required customisations we were finally ready for some actual content modeling- developing a semi-formal description of the content requirements in a form that can be used by both people and computers. A simple example of such rule could: "any printed book must have 1 title and optionally have 1 subtitle".

5.1. Modeling in XSD and importing

Initially we had an empty content model and a lot of rules and content types to describe. It turned out that a lot of these rules could be expressed as W3C Schema constraints, so we started the modelling by:

1. Creating an XML sample- usually from an exiting product
2. generating XSD from that (using OxygenXML) and refining that XSD manually
3. Importing the XSD and XML into the wiki using a bespoke XSL script to convert XML/XSD into wiki markup (for example the name of the root element would be the name of the wiki page)
4. The import process triggers the regeneration of the RDF dump and that RDF is exported to the triple store.

This process was a tradeoff as it didn't allow to model rules that can't be expressed in XSD 1.0 (for example maximum character count), but it allowed us to populate the wiki quickly with test data to develop and test features and even other systems and applications.

To automate the process, we have developed XProc to combine the above steps into a single, configurable and executable pipeline.

The pipeline is launched from a PHP upload page, so that user can specify files using a simple form rather than having to use the command line.

5.2. Modeling in the wiki

After the initial modeling, we focused to be able to do some basic modeling tasks on the wiki itself- this better reflected the typical scenarios with content model. After the initial development, usually only minor adjustments are needed or there is a need to develop to a variant of the content model based on an existing content model. For that we developed the following ways of editing the content model:

Creating a new element page (mapping to an XML element) with a form

Editing an existing element/page with that same form

A working proof of concept of a full-blown XForms based XSD editor running in a browser, inside the wiki. The editor has the capability of constructing arbitrary XML tree using current element/page as root. Upon saving, that is transformed back into wiki pages and subsequently into RDF using the same pipelines used for importing XSD/XML.

To sum up the modeling part, while there was a considerable amount of effort to set up the software stack, the core activity of content modeling is pretty trivial and using the wiki, does not even expose the user to XML or RDF per se- all the modelling activity can be done using a few buttons in the wiki.

This in no way prevents the model from being factually incorrect, but it does remove the technical barriers to keeping the model up to date and reduces the amount of time to respond to changing business requirements.

5.3. Exporting to other formats

Other than modeling and documentation, one of the main goals for the project was to use it as a canonical source from which outputs and software is generated. To enable that, every element/page has the ability to export to all supported formats (XML Sample, XSD, JSON, raw HTML, and HTML+RDFa).

Some examples of where it is useful: export to HTML to populate the website (just add CSS), export to JSON, for JavaScript-based application (js will then use the JSON files as it's data model)

5.4. Generating apps directly from the model

One of the more exciting features of the content model is the ability to generate custom editing experiences based on the content type in the model.

There is sufficient amount of information in the model to be able to generate a data-driven editor with the usability appropriate for a given content type.

For example at book or chapter level, we can generate a content planner, which is not very good for writing whole books, but gives a very good high level overview of a book and makes it easy to plan it. On the other end you have editor for individual sections where a form based editor may be more appropriate.

To prove the idea in practice we have developed exactly that: a high level content planner tool and a low level activity editing tool.

The activity tool is using XForms, specifically Betterforms.

1. Upon launching, the editor launcher interrogates the parameter to see if an editor for that element already exists.
2. If the editor for the element does not yet exist, the launcher will redirect to editor generator.
3. The generator queries the SPARQL endpoint for XSD for a given element and generates XForm from that.
4. The XForm can be tweaked manually for better UX. On subsequent run the XForm will not be overwritten as per step 2.
5. Finally the editor launches, using XForm to specify behavior and opens the XML file provided as a parameter.
6. The editor saves locally to eXistDB. eXistDB REST API is used to interface with 3rd party systems and other editors.

Being able to generate a customised editing experience by dynamically querying the content model is a very novel approach, but looking at the pace of change on the Internet is proving very useful.

An example of that is the need to migrate from Adobe Flash based applications to HTML5. Had the model been stored alongside with the Flash application, it would have to be retired or done from scratch or migrated somehow (this requiring someone with knowledge of both Flash and HTML5). With the Abstract Content Model approach, both the Flash and HTML5 are generated from the neutral and abstract content model. Because both app models are generated from the same RDF source, it is much easier to specify the migration patterns. In addition the people developing the HTML5 based apps do not need to know about Flash or even XML and RDF. The most likely route for a Web Developer to interface with the Abstract Content Model is via HTML and JSON outputs.

Such setup is also allowing to apply the (good) practices of agile development to content modeling and make the content modeling an internal part of agile software development sprints. Content model can be dynamically updated and respond quickly to changing requirements.

6. Conclusion

To conclude, the software development world is one of perpetual change. We mustn't oppose the change, it is inevitable and we should embrace and prepare for it.

By abstracting the content model and using Semantic MediaWiki as a mechanism to store it and produce multiple, often unforeseen representations, we future proof and make more valuable the work that has gone in the development of the content model, its documentation, samples and systems around it.

If you are tasked with content modeling in any capacity, which in 2014 and beyond should be a growingly popular, sought after and profitable activity, I strongly recommend that you abstract you content modeling effort and use Semantic Media Wiki or a similar solution to be able to produce multiple representations of the content model, both now and in the future.

Huge thanks and credits go to: Dr. Alex Greig Muir who was essential in developing this functionality on behalf of Kode1100 Ltd and team of Vincente Aguilar, Tomas Gradin and Albert Hervas Pi who developed a lot of custom functionality for the OHIM project.

Dr. Alex Greig Muir
: Alex was essential in developing this functionality on behalf of Kode1100 Ltd and wrote a lot of the code required to make this work.

Vincente Aguilar, Tomas Gradin and Albert Hervas Pi
: These brave men, along with yours truly, in spite of bureaucracy and distractingly good weather and food developed a lot of custom functionality for the OHIM project, which served as a great learning exercise for subsequent abstract content modeling projects.

(Semantic) MediaWiki developer community
: Likewise big thanks go to the broader Media Wiki and Semantic Media Wiki community for their ongoing contributions to the open source project.

XML Community
: Last but not least huge tanks go to the XML community which made all this possible. As many of you know, I have been a vigorous attendee of many XML events. I came for the technology and stayed for the community!

Bibliography

[1] Mark Barrenechea. Forbes. http://www.forbes.com/sites/ciocentral/2013/02/04/big-data-big-hype/

JSON and XML: a new perspective

Eric van der Vlist

Dyomedea

Abstract

A lot has already been said about the tumultuous relationship between JSON and XML. A number of binding tools have been proposed. Extensions to the XPath Data Model (XDM) and functions are being considered for XSLT 3.0 and XQuery 3.1 to define maps and arrays, two item types that would facilitate the import of JSON objects.

The author of this paper has already published and presented papers proposing an XML serialization for XSLT 3.0 maps and arrays, a detailed comparison between XML and JSON data models and a proposal to extend the XDM to better bridge the gap between these data models.

None of these efforts seems to be totally satisfying to eliminate the fundamental impedance mismatch between JSON and XML suggesting that we may not have found the right angle to look at this problem.

Rather than proposing yet another conversion methodology, this paper proposes a new perspective to look at the differences between JSON and XML which might more constructive than the ones which had been adopted so far.

1. State of the art

It is now admitted, even among the most traditionalist XML communities, that the ability to seamlessly integrate JSON and HTML5 is key to the future of the XML ecosystem.

When a business object is serialized in XML as:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<anvil reference="acme-5103">
  <weight unit="pound">9.5</weight>
  <composition>best wrought iron</composition>
  <price currency="USD">.15</price>
</anvil>
```

the underlying XML data model is usually not significant and that raises endless questions such as "should we use attributes or elements for the reference? the weight?, the composition? the price?".

Note

It must be noted that there is not such a thing as "the XML data model" and this paper will refer to the XDM 2.0. The differences between the XDM versions can be found in my paper for Balisage 2012.

JSON having been designed as the syntax to define literal object data structure in JavaScript you don't have to answer to these questions to serialize the same object in JSON:

```json
{
  "anvil": {
    "reference": "acme-5103",
    "weight": {
      "unit": "pound",
      "value": 9.5
    },
    "composition": "best wrought iron",
    "price": {
      "currency": "USD",
      "value": .15
    }
  }
}
```

When we look at these two documents with the model of the object that's been serialized in mind the translation between the two formats seems quite obvious and most of the time it really is obvious.

The simplicity of such approaches have led to a number of software solutions that perform this kind of translations.

The trickiest questions they have to solve are:

- Type inference (XML > JSON)
- Distinction between singletons (single element arrays) and primitive types (XML > JSON)
- Distinction between elements and attributes (JSON > XML).

These approaches are working fine in many cases to convert data representations back and/or forward between JSON and XML, however this is not enough when the goal is to provide an automatic conversion of any JSON data structure:

- Some implicit or explicit knowledge of the business data model serialized in the document is needed.

doi:10.14337/XMLLondon14.Vlist01

- The "lexical space" of JSON is wider than the lexical space of XML. This is the case for any string but we can't do much to cope with this difference and also for JSON keys which are commonly matched to element and attribute names.

A solution to avoid these issues is to extend the XML data model to add JSON maps and arrays to the existing item types. Early Working Drafts of XSLT 3.0 and XQuery 3.1 are both following this approach. These item types are added to the XPath Data Model (XDM) and coexist with the existing item types. The new item types being designed to be a superset of JSON maps and array it is obviously be possible to consider any JSON object as an XDM first class object.

This does not address the conversion of XML per see into JSON (considered as out of its scope) and creates a clear distinction between XML nodes and the new item types which cannot be accessed by XPath axis and require their specific sets of functions.

To avoid this segmentation of item types, several proposals have been made which extend the notion of XML elements to make them compatible with JSON maps and arrays. However it is much harder to change existing XDM item types than add new ones and these proposals are personal initiatives which are not likely to influence any standard.

Another option is to serialize JSON objects in XML and this is the approach proposed by χίμαιραλ (chimeral) and XSLT 3.0. Such serializations can provide JSON to XML round-tripping but are always verbose.

Note

You'll find a detailed list and classification of some of these solutions in my talk at XML Prague 2013

2. A new perspective

I have been working on the relationship between JSON and XML since XML Prague 2012. I have read a number of papers, listened to many presentations and done presentations on the topic at Balisage 2012 and XML Prague 2013.

Even if clever practical solutions have been proposed which work well for real world applications I still felt some itchiness for a topic both so simple and so complicated and I think that what we need is a new way to look at both formats.

This new perspective will not solve the problem by itself but might be a ground on which new proposals can be built and the purpose of this paper is not to propose a conversion between JSON and XML but to explore the relationship between these formats.

Let's take the comparison published in my talk at XML Prague 2013:

> We have both maps of key/value pairs and ordered arrays in JSON and XML, but that's where the similarities stop!
>
> In JSON, maps of key/value pairs are called objects. Keys can be any string and values may be either primitive or structured types.
>
> In XML, elements have a map of key/value pairs among their properties. This map is called attributes. Keys (i.e. attribute names) are subject to lexical restrictions and values cannot be structured types.
>
> In JSON, ordered arrays are called arrays and their members may be either primitive or structured.
>
> In XML, elements have an ordered array of children nodes among their properties. Their members can be elements, comments, PIs or text nodes. XML text nodes are the kind of nodes which is the more similar to JSON's primitive types. However, adjacent text nodes are concatenated (which of course is not the case of adjacent primitive values in a JSON array).

We are comparing on one side a data model composed of maps, arrays and atomic types and on the other side a data model composed of more complex objects, themselves including maps and arrays.

Is that really a good idea to put on the same ground basic building blocks and components made with these building blocks?

Shouldn't we rather consider XML nodes as assembly of JSON maps and arrays?

What's happening if we try to serialize the XML data model in JSON?

The XDM defines an awful lot of properties and we will focus on the main ones.

As a first approximation, an XML element can be considered as a map with:

- A name
- A map of attributes (which must be atomic types)
- An array of children (which can be either strings (text nodes) or elements.

Serialized with these conventions, the XML document describing the anvil can be seen as:

```
{
  "name": "anvil",
  "attributes": {"reference": "acme-5103"},
  "children": [
    {
      "name": "weight",
      "attributes": {"unit": "pound"},
      "children": ["9.5"]
    },
    {
      "name": "composition",
      "attributes": {},
      "children": ["best wrought iron"]
    },
    {
      "name": "price",
      "attributes": {"currency": "USD"},
      "children": [".15"]
    }
  ]
}
```

The difference that immediately strikes the eyes when compared to the previous JSON definition of the anvil is that it is more verbose and that's true but this is so by design: this is no longer a JSON description of an anvil but a JSON description of the XML document describing an anvil.

This description is very close to the XDM and can be used with mixed content:

```
<p>I can support
  <a href="http://en.wikipedia.org/wiki/PCDATA">
    <b>mixed</b> content</a>!</p>
```

becomes:

```
{
  "name": "p",
  "attributes": {},
  "children": [
    "I can support ", {
      "name": "a",
      "attributes": {
        "href":
        "http://en.wikipedia.org/wiki/PCDATA"
      },
      "children": [ {
          "name": "b",
          "attributes": {},
          "children": ["mixed"]
        },
        " content"
      ]
    },
    "!"
  ]
}
```

3. So what?

3.1. A good theory

This exercise clearly shows the relation between JSON and XML:

- JSON is a generic format to describe data structures made of maps, arrays and a few basic simple types.
- XML is a more specialized format to describe tree composed of "nodes" which do carry their own semantic and can be serialized in JSON.

This difference is the essence of the "fat" described by Douglas Crockford in his famous "Fat-Free alternative to XML" paper at XML 2006 and might have been widely acknowledged by the XML community if presented in a less inflamed fashion.

The questions raised when trying to convert our XML and JSON samples such as:

- from XML to JSON: is the weight a property of the anvil? can it occur more than once and be included in an array?, ...
- from JSON to XML: should the composition be an element or an attribute? , ...

are a sign that we are misusing XML and using its nodes as generic data structure components which they are not meant to be.

3.2. Navigation

Of course, acknowledging this fundamental difference between JSON and XML doesn't solve the issue of converting XML to JSON (and vice versa) and does not even diminish the need for such conversions.

That doesn't make this serialization a purely theoretical exercise deprived of any practical use and XML documents and serialized XML documents are in fact quite handy to manipulate in JavaScript!

3.2.1. In plain JavaScript

Navigating amongst attributes is of course really easy:

```
anvil.attributes["reference"]
```

or when the attribute name is a valid JavaScript name:

```
anvil.attributes.reference
```

Navigating amongst children elements may seem more challenging but would arguably not be more difficult than doing so using the DOM... However it becomes much easier if we define a simple function such as:

```
elt=function(name) {
  return function(o) {
    return o.name==name
  }
}
```

and use it to filter children:

```
anvil.children.filter(
  elt('price')
)[0].attributes.currency
```

This can be still easier if we append a new method to JavaScript objects:

```
Object.prototype.elt = function(name) {
  return this.filter(elt(name))
}
```

and just use it to access children elements:

```
anvil.children.elt('price')[0].attributes.currency
```

3.2.2. With a JSON query library

It would be easy to add other methods to facilitate this navigation but we can also rely on existing libraries such as json:select() which inspiration is to be the jQuery of JSON objects.

Let's start by defining a shorter name for json:select()'s match() method:

```
js=JSONSelect.match
```

If we feel lucky we can access the anvil's reference like this:

```
js('.reference', anvil)
```

This would work with our simple example but like jQuery, json:select() does deep searches and this relies on the fact that there is no reference attribute anywhere else than for the anvil root element. A safer method is to restrict our expression to match only the root:

```
js(':root > .attributes > .reference', anvil)
```

Again, this would work with our simple example because the root element is the anvil. If we want to find the references of any anvil element that might be anywhere in the JSON object we should be less restrictive and write:

```
js(':has(.name:val("anvil")) > .attributes > .reference', anvil)
```

which can be (slightly) shortened into the almost equivalent:

```
js(
  '.name:val("anvil") ~ .attributes > .reference',
  anvil
).toString()
```

This is more verbose than the `anvil.attributes["reference"]` that we've written in plain JavaScript but that does more since it's performing a deep search for any anvil. This is also quite generic and the same pattern can be used to search for price's currencies:

```
js(
  '.name:val("price") ~ .attributes > .currency',
  anvil
)
```

3.3. Other usages

The best indication that this kind of serialization may be useful is that... it is not new and has already be implemented for several kind of applications!

3.3.1. html2json

Developed in October 2011, html2json is very similar the serialization used in this paper: the only differences are in names (tag instead of name, attr instead of attributes and child instead of children).

Unfortunately, its author hasn't documented its use cases and we don't really know what his JavaScript implementation has been used for.

3.3.2. JsonML

JsonML has its own website which is much more explicit about its motivation and use cases:

> JsonML (JSON Markup Language) is an application of the JSON (JavaScript Object Notation) format. The purpose of JsonML is to provide a compact format for transporting XML-based markup as JSON which allows it to be losslessly converted back to its original form.
>
> Native XML/XHTML doesn't sit well embedded in JavaScript. When XHTML is stored in script it must be properly encoded as an opaque string. JsonML allows easy manipulation of the markup in script before completely rehydrating back to the original form.
>
> --JsonML.org

The design choices behind JsonML seem to have been drawn by this use case and bend the result toward something which is concise and reasonably easy to read: XML elements are represented by an array of a string representing the element name, an optional map for its attributes and an optional children elements which can be either strings or arrays. With these conventions, the serialization of the anvil is:

```
[
  "anvil",
  {"reference": "acme-5103"},
  [
    "weight",
    {"unit": "pound"},
    "9.5"
  ],
  [
    "composition",
    "best wrought iron"
  ],
  [
    "price",
    {"currency": "USD"},
    ".15"
  ],
]
```

While this is more concise than the serializations that we've seen so far, I find it much less natural to convert elements into arrays than into maps.

Beside this first use case as a serialization to transport XML as JSON, JsonML.org mentions browser side templating (JSBT) as a second use case.

3.3.3. fastFrag

Client-side templating is the main use case of fastFrag which proposes its own serialization. This serialization is basically similar to what I propose in this paper with some added features to make it easier to use to create HTML fragments:

- The name is optional. Called type it defaults to "div".
- The most common attributes (id, css, ...) have been "promoted" to appear directly as keys of top level elements objects instead of being placed in the attributes map.
- The "attributes" key can also be spelled "attrs" or "attr" for brevity.
- A "text" key has been added which is a shortcut to create text only elements.

If we forget these features and the fact that FastFrag is not meant to be used with arbitrary XML we get a serialization which is very similar to what we've already seen:

```
{
  "attrs": {"reference": "acme-5103"},
  "content": [
    {
      "attrs": {"unit": "pound"},
      "content": {"text": "9.5"},
      "type": "weight"
    },
    {
      "content": {"text": "best wrought iron"},
      "type": "composition"
    },
    {
      "attrs": {"currency": "USD"},
      "content": {"text": ".15"},
      "type": "price"
    }
  ],
  "type": "anvil"
}
```

4. Conclusion

JavaScript has become the assembly language for programming languages on the web.

I have no doubt that JSON should be the assembly language for any data model -including XML- on the web and we have seen how easy it is to use JSON as a serialization format for a subset of the XDM.

This presentation is too short to show how this subset could be extended to support other XDM item types and properties such as namespaces, comments, processing instructions but I have no doubt that this could be done in a number of different ways without altering too much the simplicity of the serialization.

Beside the ability to round-trip XML to JSON conversions defining such a serialization would be a good way to document the XML data model and could be a basis for building XML libraries on top of existing JSON libraries.

Understanding that JSON is lower level than XML also helps to understand why round-tripping XML to JSON is simple while round-tripping JSON to XML is harder and verbose or application dependent.

The simplistic representation shown in this paper is merely a first step and defining a standard representation of the XDM in JSON would help to acknowledge this relationship and to implement tools to facilitate a peaceful coexistence of both formats.

Charles Foster

XML London 2014
Conference Proceedings

Published by
XML London

103 High Street
Evesham
WR11 4DN
UK

This document was created by transforming original DocBook XML sources into an XHTML document which was subsequently rendered into a PDF by **Antenna House Formatter**.

1st edition

London 2014

ISBN 978-0-9926471-1-7