XML Prague 2018

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 8–10, 2018

XML Prague 2018 – Conference Proceedings
Copyright © 2018 Jiří Kosek

ISBN 978-80-906259-4-5 (pdf)
ISBN 978-80-906259-5-2 (ePub)

X-definition

X-definition

A comprehensible description of XML documents

The X-definition is an XML document that can be used to describe a set of XML documents.

- Easy to design and to understand
- Can be used to describe and to process JSON data
- The comprehensibility of the X-definition source makes it extremely easy to create a documentation of XML structures.

Validation, processing or construction

The X-definition integrates the validation process of XML data, the processing and the construction of a new XML document - all in the same language.

- An easy way to facilitate the maintenance of large projects
- A generation of a detailed error report
- Tools for a dynamic error processing

The Java environment interconnection

The X-definition allows Java projects interconnections.

- It is possible to execute Java methods and to access Java objects
- The feature "X-Components" enables to generate source Java code representing XML structure and to use it in Java programs (in a similar way as JAXB)

Connection to databases and external data

In the X-definition you can execute database statements (either in relational databases or in XML databases). It is possible also to access external data.

An easy maintenance of large projects

An excessive collection of X-definitions has been successfully used in real projects in which very often large data files needed to be processed (many GB).

syntea

www.xdef.cz | www.syntea.cz

Table of Contents

General Information

Date

February 8th, 9th and 10th, 2018

Location

University of Economics, Prague (UEP)
nám. W. Churchilla 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Program Committee

Produced By

XML Prague, z.s. (http://xmlprague.cz/about)
Faculty of Informatics and Statistics, UEP (http://fis.vse.cz)

Sponsors

oXygen (http://www.oxygenxml.com)
le-tex publishing services (http://www.le-tex.de/en/)
Antenna House (http://www.antennahouse.com/)
Saxonica (http://www.saxonica.com/)
speedata (http://www.speedata.de/)
Syntea (http://www.syntea.cz/)

<oXygen/> xml editor

le-tex publishing services

A Data Usability Company ANTENNA HOUSE

SAXONICA.COM XSLT AND XQUERY PROCESSING

speedata

syntea

Preface

This publication contains papers presented during the XML Prague 2018 conference.

In its thirteenth year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup and semantic on the Web, publishing and digital books, XML technologies for Big Data and recent advances in XML technologies. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 8–10 February 2018 at the campus of University of Economics in Prague. XML Prague 2018 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, University of Economics in Prague.

The full program of the conference is broadcasted over the Internet (see http://xmlprague.cz)—allowing XML fans, from around the world, to participate on-line.

The Thursday runs in an unconference style which provides space for various XML community meetings in parallel tracks. Friday and Saturday are devoted to classical single-track format and papers from these days are published in the proceeedings. Additionally, we coordinate, support and provide space for XProc working group meeting collocated with XML Prague.

We hope that you enjoy XML Prague 2018 – especially as this is a very special edition – the last day of the conference is 20th anniversary of XML Recommendation publication.

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

Assisted Structured Authoring using Conditional Random Fields

Bert Willems
FontoXML
<bert.willems@fontoxml.com>

Abstract

Authoring structured content with rich semantic markup is repetitive, time consuming and error-prone. Many Subject Matter Experts (SMEs) struggle with the task of applying the correct markup. This paper proposes a mechanism to partially automate this using Conditional Random Fields (CRF), a machine learning algorithm. It also proposes an architecture on how to continuously improve the CRF model in production using a feedback loop.

Keywords: XML, Conditional Random Fields, Structured Authoring, Machine Learning

1. Introduction

With the increasing adoption of structured XML content, the amount of work required from Subject Matter Experts (SMEs) increases. Not only are they required to capture their knowledge as information to others, they are increasingly asked, and sometimes even required, to mark up the information with the appropriate semantic and structural metadata in the form of XML tags and attributes.

Examples of those markup tasks include:

- Structuring bibliographic references to tag authors, journal name, publisher etc.
- Marking up tasks, not with ordered lists but with steps.
- Marking up interactive questions, like multiple choice questions.

Although WYSIWYG XML editors help to make this task as easy as possible, the fact remains that there is additional work to be done that is often repetitive and error-prone. FontoXML conducted multiple studies to determine whether the effort of manual tagging affected adoption. The results showed a consistent negative effect: SMEs and their editorial colleagues are hesitant to adopting structured authoring. In some cases this meant reverting back to their unstructured content processes, leading to unrealized potential.

Prior implementations, like GROBID [3], apply markup automatically. This paper proposes to introduce Machine Learning (ML) to the authoring process instead. The reason for this is the inaccuracy of the state-of-the-art ML algorithms: like humans, they make mistakes. Allowing SMEs to (correct and) accept a machine provided suggestion will result in a more accurate markup. Furthermore, this approach allows for the creation of a feedback loop, allowing the machine to improve over time.

This paper focuses on the task of structuring bibliographic citations, although the proposed architecture scales to many of the tasks required for properly structured content.

2. Model

This section describes the model used for recognizing bibliographic citations and extracting the relevant labels from it. The model used in this paper follows a divide-and-conquer strategy and is made up out of two separate models: The Citation Model and the Name Model. Partial results from the Citation Model cascade into the Name model to more detailed results.

2.1. Citation Model

The goal of the Citation Model is to classify a sequence of text with tags that make up the parts of the citation. The tags are derived from the TEI P5 vocabulary [12] and are encoded using the IOB tagging scheme [8].

The following tags are distinguished:

- author
- orgName
- editor
- publisher
- pubPlace
- date
- idno (bibliographic identifier)
- analytic (articles, poems, etc.)
- monographic (books, single & multi volumes, etc.)
- journal
- series
- unpublished
- volume
- issue
- pages
- chapter

For example, the sequence *Erickson, T. & Kellogg, W. A. "Social Translucence: An Approach to Designing Systems that Mesh with Social Processes." In Transactions on*

Computer-Human Interaction. Vol. 7, No. 1, pp 59-83. New York: ACM Press, 2000. is tagged as[1]:

author	Erickson, T. & Kellogg, W. A.
analytic	Social Translucence: An Approach to Designing Systems that Mesh with Social Processes
journal	Transactions on Computer-Human Interaction
volume	7
pages	59-83
pubPlace	New York
publisher	ACM Press
date	2000

2.2. Name Model

The Name Model is much simpler[2] compared to the Citation Model. The purpose of the Name Model is to distinguish names in a given sequence of text. To be more specific in the *author* and *editor* labels cascading from the Citation Model. The tags are also encoded using the IOB tagging scheme.

The following tags are distinguished:

- forename
- middlename
- surname

For example, the sequence *Erickson, T. & Kellogg, W. A.* is tagged as:

surname	Erickson
forename	T
surname	Kellogg
forename	W.
middlename	A.

[1]The IOB prefixes are combined and the outside tags are removed in the example output for brevity.
[2]Although the model is comparatively simple, handling names is surprisingly complex. See: http://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/ . The point of the model is to learn from examples in the training data rather than manually coded rules.

2.3. Linear Chain Conditional Random Field

Both the Citation and the Name model are implemented using the Conditional Random Fields (CRF) algorithm [7]. According to the research performed by Fuchun Peng and Andrew McCallum [10], CRFs work well for extracting structured bibliographic citations from research papers, achieving good performance. This algorithm is also used in similar implementations like GROBID and MALLET [9].

To get a sense of how CRFs work, consider the sentence “I’m at home.”. Now consider the sentence “I’m at kwaak.”. Based on both sentences one intuitively understands that “kwaak” is some sort of location because we know that “home” is also a location and the words appear in the same *context*.

CRFs take into account the context in which a word appears and some other features like “is the text made up out of numbers?”. More precisely: an input sequence of observed variables X represents a sequence of observations (the words with the associated features which make up a sentence) and Y represents a hidden (or unknown) state variable that needs to be inferred given the observations (the labels). The Y_i are structured to form a chain, with an edge between each $Y_{(i-1)}$ and Y_i. As well as having a simple interpretation of the Y_i as "labels" for each element in the input sequence, this layout admits efficient algorithms for:

1. model training, learning the conditional distributions between the Y_i and feature functions from some corpus of training data.
2. decoding, determining the probability of a given label sequence Y given X.
3. inference, determining the most likely label sequence Y given X.

For a more detailed introduction to CRFs, see An Introduction to Conditional Random Fields for Relational Learning [11].

2.4. Implementation

For the implementation of the CRFs, an implementation based on CRFSharp [1] is used. CRFSharp is a .NET Framework 4.0 implementation of Conditional Random Fields written in C#. Its main algorithm is similar with CRF++ written by Taku Kudo [6]. It encodes model parameters by L-BFGS. Moreover, it has many significant improvements over CRF++, such as totally parallel encoding and optimized memory usage. The CRFSharp implementation was modified to target the .NET Standard 2.0 to allow cross-platform usage in .NET Core applications.

2.5. Training

In order for the CRFs to learn the desired outcomes, they must be trained first using correctly tagged example inputs. For the training of the Citation Model, a corpus of 439 TEI documents is used originating from the GROBID GIT repository. The corpus contains 10.288 properly marked up bibliographic citations (`<bibl/>`). 6.467 citations are used for training, while the remaining 3.821 are used for evaluation. For the training of the Name Model, a corpus of 13 TEI documents is used originating from the GROBID GIT repository. The corpus contains 403 properly marked up names of which 370 names are used for training, and the remaining 33 names are used for evaluation.

Both models were trained with a maximum iteration count of 1000 using L2 regularization running on 8 threads in parallel. The training of the Name Model takes no more than a minute while training the Citation Model takes around 2,5 hours on a laptop with an Intel i7-4702HQ processor and 16GB of RAM. Training was clearly CPU bound; the 8 logical processors were 100% utilized. Memory usage was around 1.5 GB. Training speed can possibly be improved using the GPU rather than the CPU. However this was not explored.

2.6. Model Evaluation

The trained models are evaluated against previously unseen examples. Both models are scored on the overall performance of all their labels. Both models are scored using the accuracy and F_1 metrics which are common scoring metrics for Named Entity Recognition (NER) models. The scoring is defined as:

1. Accuracy; Where TP is the number of true positives, FP is the number of false positives, TN is the number of true negatives and FN the number of false negatives. Accuracy is calculated as: $accuracy = \frac{TP + TN}{TP + FP + TN + FP}$.
2. F1 measure; Precision, recall and F1 measure are defined as follows: $precision = \frac{TP}{TP + FP}$, $recall = \frac{TP}{TP + FN}$, $F_1 = 2 \times \frac{precision \times recall}{precision + recall}$.

Both models perform reasonably well on the evaluation dataset. Table 1 shows the results of the evaluation of both models:

Table 1. Evaluation Results

Model name	**Accuracy**	**F_1**
Citation Model	99,53%	99,72%
Name Model	99,36%	99,31%

One of the observations made during the evaluation is that the Citation Model has a hard time predicting the correct type for the titles (analytic vs monographic and journal vs series vs unpublished). Similarly, it has a hard time distinguishing between author and editor. This is partly due to the local nature of CRFs; the CRFs in this application use a context window of 4 words around the target word. Introducing a bidirectional Long Short-Term Memory (LSTM) in combination with a CRF layer overcomes this local nature [13]. Another boost may be a gained from introducing lexicons for journal titles etc.

2.7. XML Mapping

Mapping the extracted labels to an XML vocabulary is straightforward, see Table 2 for a mapping between the labels and the JATS [5] & TEI vocabularies:

Table 2. Labels to XML mapping

Label	**JATS `<mixed-citation/>`**	**TEI `<bibl/>`**
author	`<name/>`	`<author/>`
orgName	`<collab/>`	`<orgName/>`
editor	`<person-group person-group-type="editor"/>`	`<editor/>`
publisher	`<publisher-name/>`	`<publisher/>`
pubPlace	`<publisher-loc/>`	`<pubPlace/>`
date	`<year/>`	`<date/>`
idno	`<object-id/>`	`<idno/>`
analytic	`<article-title/>`	`<title level="a"/>`
monographic	`<article-title/>`	`<title level="m"/>`
journal	`<source/>`	`<title level="j"/>`
series	`<source/>`	`<title level="s"/>`
unpublished	`<source/>`	`<title level="u"/>`
volume	`<volume/>`	`<biblScope unit="volume"/>`
issue	`<issue/>`	`<biblScope unit="issue"/>`
pages	`<page-range/>`	`<biblScope unit="pages"/>`
chapter	`<chapter-title/>`	`<biblScope unit="chapter"/>`

An interesting observation is that the model is not specific to a particular vocabulary. This means training data can be in one vocabulary while the results of the model are represented in another vocabulary. Also, a mix of vocabularies can be used as training data.

Both the JATS and the TEI vocabularies have specific tags for marking up a bibliographic citation. Both container elements allow character data and do not impose any ordering restrictions. JATS also offers the non-mixed `<element-citation/>` element which provides more structure. In case of a vocabulary that does impose structure, the mapping can be implemented with XML Conditional Random Fields (XCRFs) [2]. This CRF implementation predicts labels of tree nodes instead of labels for text.

3. The Human Factor

Although the model used performs quite well, it will still make mistakes. Rather than programmatically marking up the recognized citation in the XML document, this paper proposes the treat them as suggestions. A user interface for the SMEs allow them to confirm and correct the citations before marking them up in the XML document. This section describes some of the common errors generated by the model, a user interface to correct them and a mechanism for the system to learn the manually made corrections.

3.1. Analysis of Model Errors

As indicated in the section Section 2.6, the Citation Model has a hard time distinguishing between certain labels. A further statistical analysis on the evaluation errors proves this assumption. Table 3 contains the top 10 errors (about 60% of the total errors):

Table 3. Evaluation Errors

#	**Frequency**	**Actual**	**Expected**
1	129	analytic	monographic
2	60	series	journal
3	45	monographic	analytic
4	37	monographic	O[a]
5	31	analytic	journal
6	29	O	idno
7	18	monographic	journal
8	18	journal	monographic
9	18	author	editor
10	18	journal	author

[a]The *O*utside label of the IOB tagging scheme, indicates absence of a label.

This table reveals three classes of issues:

1. Distinguish between related labels, for example analytic and monographic occur in the same location. This is the cause of results 1, 2, 3, 5, 7, 8, 9, 10.
2. Distinguish whether certain characters are inside a title or not. This is the cause of result 4.
3. Distinguish identifiers, more specifically URLs. This is the cause of result 6.

The first class of issues is caused by labels that are closely related to each other. Intuitively it makes sense that it is hard to determine the difference between article and book titles, even for humans. The range however is correctly determined in all those cases. Meaning, changing the label would suffice for the user to fix the mistake.

The second class of issues is caused by inconsistencies in the evaluation data. Some titles include a dot while it has been omitted from others. This is solved with a post processing rule either consistently placing the dot inside or outside the title. Alternatively, the training data may be updated.

The third class of issues is caused by the model not understanding URLs. This is best solved by adding a URL feature to the model and retraining it.

In conclusion, it is clear that fast majority of issues can be addressed by simply allowing the SME to select a different, but closely related label.

3.2. User Interactions

Based on the analysis in the previous section, we can formulate one requirement: As a SME I can correct the type of title/person in a citation. This requirement is implemented using two UI patterns:

1. A (context) menu options to toggle between the types.
2. A form with dropdown lists to select different types.

The benefit of the first pattern is that it will also work for correcting citations that have been manually put into the system; it works after the fact. Both context menu options and toolbar buttons have been implemented to allow SMEs to use their preferred method. See Figure 1 for an an example of such an UI.

The benefit of the second pattern is that it is task-specific, allowing the SME to work with greater focus. It is implemented in its own dedicated UI component allowing users to quickly validate the recognized results and correct where needed. This puts the SME in control. The dropdowns are populated with the alternatives sorted by the likelihood of the different evaluations of the model. See Figure 2 for an an example of such an UI.

There is no need to choose between the two; they can very well be combined.

3.3. Structured Content Feedback Loop

Integrating ML models in the authoring process helps to automate some of the repetitive tasks that are part of structured content authoring. As shown in the

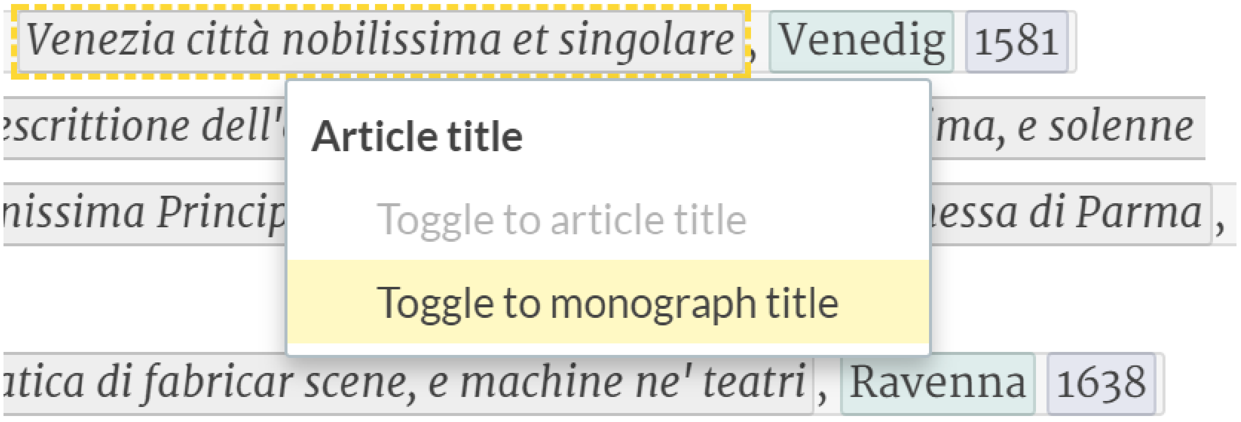

Venezia città nobilissima et singolare, Venedig 1581
escrittione dell'
ma, e solenne
nissima Princip
essa di Parma,
Article title
Toggle to article title
Toggle to monograph title
itica di fabricar scene, e machine ne' teatri, Ravenna 1638

Figure 1. A (context) menu options to toggle between the types

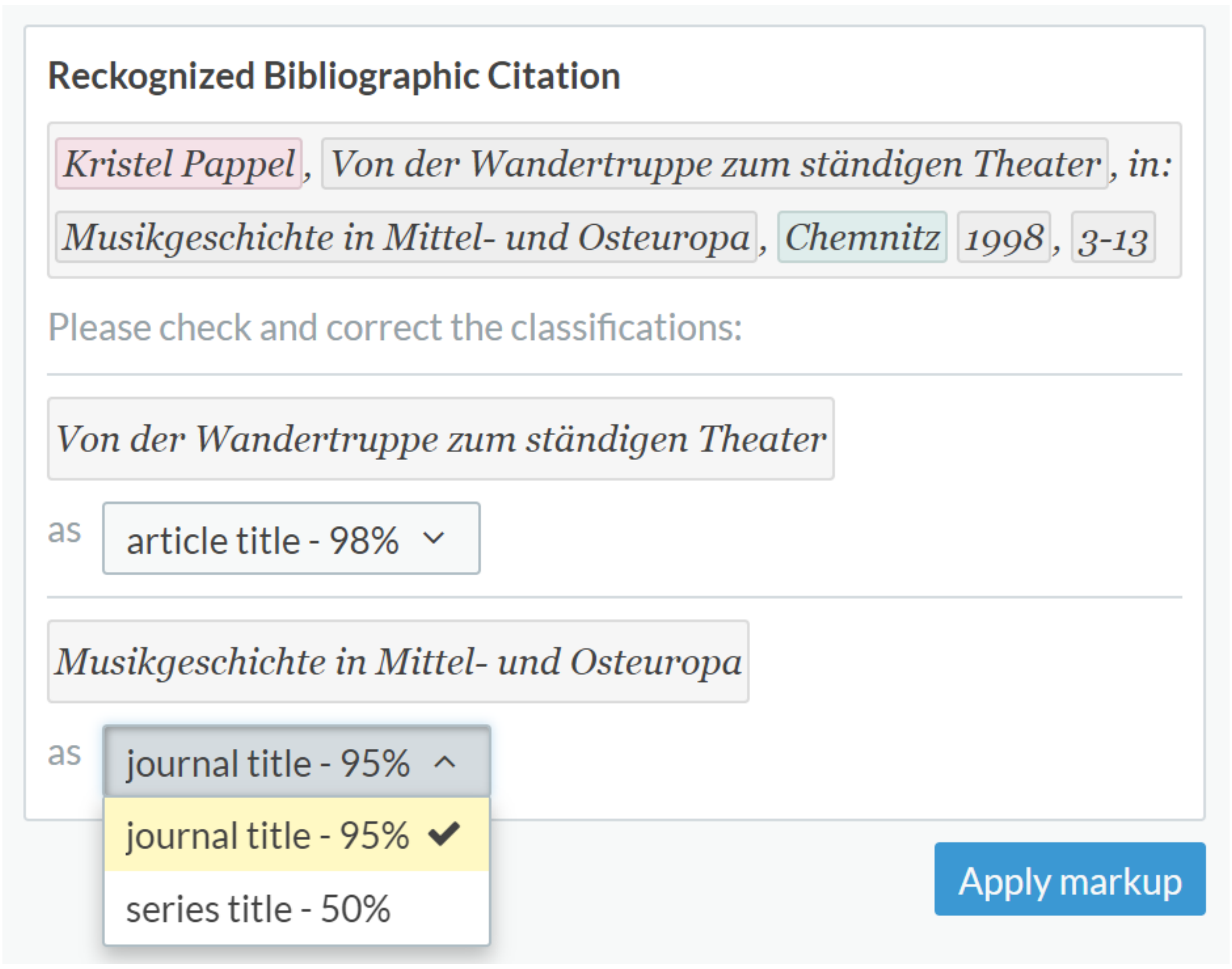

Reckognized Bibliographic Citation
Kristel Pappel, Von der Wandertruppe zum ständigen Theater, in: Musikgeschichte in Mittel- und Osteuropa, Chemnitz 1998, 3-13
Please check and correct the classifications:
Von der Wandertruppe zum ständigen Theater
as
article title - 98%
Musikgeschichte in Mittel- und Osteuropa
as
journal title - 95%
journal title - 95% ✓
series title - 50%
Apply markup

Figure 2. A form with dropdown lists to select different types

analysis of the model performance, there is room for improvement. Integrating the model in the authoring process creates an interesting opportunity to build a feedback loop between the model and the SME.

In order to implement such a loop, the system is expanded with a storage. The storage holds the training data as well as the associated metadata, like date added and user who added the training data. Each record in the training data has a unique identifier. The system is also expanded to include a queue which holds the identifiers of the records that have not been trained on yet. The editor application is modified to send all XML citations to the system upon (auto)save. The system adds the citation to the storage and enqueues it for training. At certain intervals the system retrains the model.

The system has two modes for retraining the model:

1. Incremental retraining; reuses weights from the previously trained model.
2. Full retraining; retrains from scratch.

In both modes the updated corpus (old + new) is used for training. The incremental mode is much faster but has an important limitation in the CRFSharp implementation: it can't handle new labels. The system therefore detects if new labels were added and chooses the appropriate mode. Once retraining has been completed the old model is swapped for the new model and used consecutively for generating the improved suggestions.

Retraining the model automatically imposes some risks. For example, the model may overfit if the SMEs have been working with one particular type of citation for a while. Another example is that an SME (deliberately) inserts the wrong tags throwing the model off guard. The impact of these risks hasn't been explored within the scope of this paper. The metadata, associated with each training record, allows after the fact filtering of records at the cost of a full retraining.

4. Conclusion and Further Work

Using Machine Learning, in this paper Conditional Random Fields, to (semi) automatically markup citations reduces the time SMEs need to spend applying markup manually. In cases where the model mispredicted the correct markup, in general, the correction is just changing the predicted label. Using a specific user interface to change those labels, SMEs stay control while still saving time.

Integrating ML into the authoring process not only saves valuable SME time, it also allows for a Structured Content Feedback Loop to be created. This feedback loop continuously gathers additional training data, reviewed by the SME, to improve the model. The improved model will in turn generate better predictions.

The CRF algorithm works reasonably well but has some limitations. One such limitation is the context window in which it operates, which severely limits its ability to reason on entire sentences. Another limitation is that it requires manual feature engineering. Both limitations can be addressed by implementing a much more powerful model based on bidirectional LSTM in combination with CRFs [13] [4].

Bibliography

[1] Fu, Zhongkai . 2017. CRFSharp. `https://github.com/zhongkaifu/CRFSharp`.

[2] Gilleron, Rémi, Florent Jousse, Isabelle Tellier, and Marc Tommasi. 2006. "XML Document Transformation with Conditional Random Fields." INEX.

[3] 2008-2017. GROBID. `https://github.com/kermitt2/grobid`.

[4] Huang, Zhiheng, Wei Xu, and Kai Yu. 2015. "Bidirectional LSTM-CRF Models for Sequence Tagging." CoRR abs/1508.01991.

[5] JATS Standing Committee. 2015. Journal Article Tag Suite. `https://jats.nlm.nih.gov/`.

[6] Kudo, Taku. 2017. CRF++: Yet Another CRF toolkit. `https://taku910.github.io/crfpp/`.

[7] Lafferty, John D, McCallum Andrew, and Pereira Fernando. 2001. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data." Proceedings of the International Conference on Machine Learning. 282–289.

[8] Lance A. Ramshaw, Mitchell P. Marcus. 1995. "Text Chunking using Transformation-Based Learning." ACL Third Workshop on Very Large Corpora 82-94.

[9] McCallum, Andrew Kachites. 2002. MALLET: A Machine Learning for Language Toolkit. `http://mallet.cs.umass.edu`.

[10] Peng, Fuchun, and McCallum Andrew. 2004. "Accurate Information Extraction from Research Papers using Conditional Random Fields." HLT-NAACL.

[11] Sutton, Charles A., and Andrew McCallum. 2012. "An Introduction to Conditional Random Fields." Foundations and Trends in Machine Learning 4: 267-373.

[12] TEI Consortium, eds. 2017. TEI P5: Guidelines for Electronic Text Encoding and Interchange. `http://www.tei-c.org/Guidelines/P5/`.

[13] Zhiheng, Huang, Xu Wei, and Yu Kay. 2015. "Bidirectional LSTM-CRF Models for Sequence Tagging." CoRR abs/1508.01991.

XML Success Story: Creating and Integrating Collaboration Solutions to Improve the Documentation Process

Steven Higgs
Syncro Soft
<steven_higgs@sync.ro>

Abstract

This paper discusses many of the challenges that Syncro Soft (the makers of the Oxygen XML suite of products) faced when trying to improve the collaboration part of their documentation process, and provides details about their solutions for addressing those challenges. By developing new collaboration tools, refining internal processes, and integrating creative collaboration solutions into existing applications, they found ways to effectively improve the quality of their documentation, simplify various procedures, and increase the efficiency of their entire collaboration process.

1. Introduction

Over the years, our world has become more and more reliant on mobile technologies and most companies have moved toward solutions that maximize efficiency and productivity for all areas of their business. Of course, this also applies to the documentation process. Over the last few years, we (Syncro Soft) have had many users request new features that will improve their documentation process. We are also always continually trying to improve our own process of creating and publishing documentation to make the whole process more efficient and effective. One specific area that presents a variety of challenges is the collaboration part of the process. Since challenges often present opportunities, we focused a lot of time and effort over the past few years in finding solutions to make the collaboration part of our documentation process more efficient, simple, and effective.

Every company has their own organizational structure, workflow, set of requirements, and collaboration challenges. The process usually includes various different types of collaborators who may not have access to the same applications or the same level of writing skills. Every individual has their own perspective, motivations, personality, and everyone has their own way of thinking and communicating. Collaboration often requires documents or messages to be sent back and forth numerous times, and without a fluid, dynamic system in place, this can slow down the process and decrease productivity. Many companies also have personnel spread out across multiple locations and in some cases, multiple time zones. These are just some of the many challenges that most documentation teams, including ours, face when trying to collaborate with others throughout the documentation process.

This paper focuses on a real world use-case of how we addressed these collaboration challenges by creating and integrating Oxygen products and other tools. This use-case will show how we managed to make our collaboration process more efficient, effective, flexible, and fluid for everyone involved in the documentation workflow.

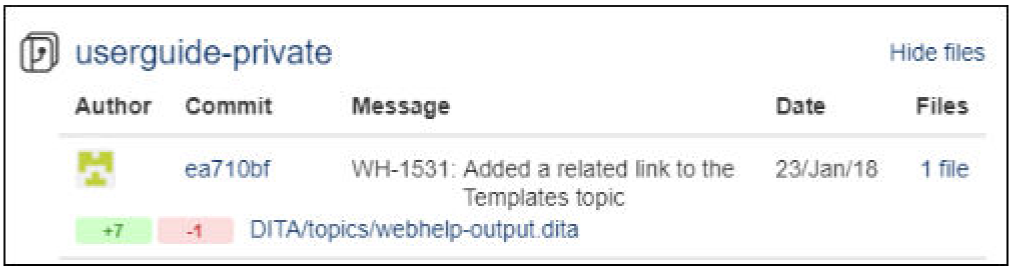
2. Customizing an Issue Tracking Application to Maximize Collaboration Efficiency for all Departments

Most companies use some sort of issue tracking application to manage tasks, projects, problems, etc. Every company also has its own way of managing and tracking issues, but this is where the documentation process often begins. An issue identifies a need for adding or editing documentation, the progress is noted and tracked, and eventually reaches a completion status. Somewhere throughout this process, collaboration is often needed and this is where some sort of customization can be helpful.

In our case, we use Atlassian JIRA for all of our issue tracking, project management, and Agile assessment. JIRA is highly customizable and there's a variety of different dashboards, various ways to filter or search for issues, and we were able to configure the workflow for each type of issue to accommodate multiple different departments. While having the ability to adjust the workflow, track, manage, and filter the issues are all very important, we needed it to go much further than that to address our collaboration challenges and make our process more efficient.

One obvious way to do this was to configure automatic notifications so that whenever an issue progresses to a new stage, the appropriate person or department receives some sort of notification. The issues are also configured so that certain fields are required to progress to the next stage. This ensures that all departments have the information they need to complete their part of the process. We also simplified and streamlined the JIRA interface by filtering certain fields, options, and possible actions so that only the necessary information is displayed. This not only makes the process easier and faster, but also helps to prevent mistakes.

Another way that we have managed to maximize the efficiency of our issue tracking process is to integrate our development and documentation repositories with our JIRA system. Developers use the JIRA issue number whenever they commit a change to our Subversion repository and the Documentation team also uses the JIRA issue number whenever they commit a change to our Git repository. The JIRA issue then displays each commit that was made for that particular issue number as a link and the details of the actual commit can be viewed by simply clicking the link.



userguide-private

Hide files

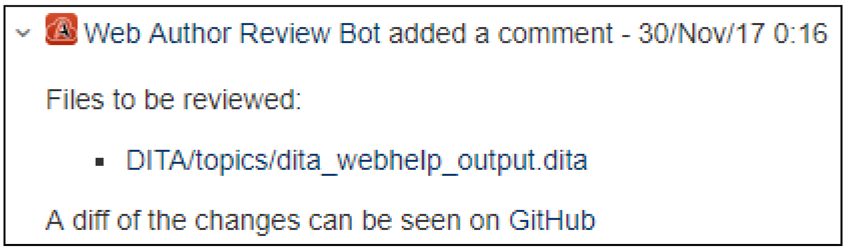
Author	Commit	Message	Date	Files
	ea710bf	WH-1531: Added a related link to the Templates topic	23/Jan/18	1 file

+7 -1 DITA/topics/webhelp-output.dita

Figure 1. Commit Link in JIRA

For issues that require an addition or change in the documentation, once our documentation team finishes their process and it progresses to the next phase, the quality assurance team and original issue creator then need to proofread the changes. When analyzing our workflow, we found that the methods used by the various people proofreading the documents were inconsistent and sometimes tedious. Our challenge was to find a way to streamline the process.

We already had an existing web-based visual editor called Oxygen XML Web Author, so we integrated something we call a *Web Author Review Bot* that displays documentation commits directly on the particular JIRA issue, and with this mechanism, clicking on the link opens the actual changed document in the web-based visual editor where they can see the changes in the context of the entire document. The advantages of this approach are that the person looking at the commit can review the changes in a visual editor, make corrections or add comments directly in the source document, commit those changes so that the documentation team simply pulls the changes to merge them into their project, and this eliminates several steps in the entire process and helps to improve accuracy and quality.



Web Author Review Bot added a comment - 30/Nov/17 0:16

Files to be reviewed:

- DITA/topics/dita_webhelp_output.dita

A diff of the changes can be seen on GitHub

Figure 2. Web Author Review Bot Link

By creatively configuring our third-party issue tracking application and finding ways to integrate our own existing applications, the entire workflow has been simplified and is far more efficient for all departments. Simplifying and improving the collaboration workflow ultimately resulted in more consistency and better documentation.

3. Developing a Collaboration Solution to Maximize Productivity and Improve Documentation Quality

One of the biggest challenges most documentation teams face when collaborating with developers, engineers, or other subject matter experts is that not only do they each have a different focus and purpose for their contributions, but in many cases they each think and communicate in very different jargons. For example, a developer may have a very technical way of expressing a new feature they've implemented, and while other highly technical users may understand their jargon, the technical writer needs to be able to translate that into a vernacular that not only a technical user will understand but also a common user that just wants to know how to use the product without caring about what's "under the hood". This usually requires a lot of communication and collaboration between the parties involved in the process and unless they happen to have the ability to collaborate in person, and the time to do so, this can present quite a challenge for the documentation team.

In our particular case, to come up with a solution for this intricate challenge, we analyzed our workflow and found that our documentation process usually looks something like this:

1. The developer writes and shares their notes about a new feature in JIRA (our third-party issue tracking application) and they indicate that it needs an addition or change in the documentation.
2. The technical writer communicates with the developer to make sure the documentation requirements are understood.
3. The technical writer documents the new feature based upon the information they have, as well as any research or testing they do themselves.
4. The technical writer sends the documentation to the developer for review.
5. The developer reviews the documents and sends their feedback to the technical writer.
6. The technical writer adjusts the documentation according to the feedback from the developer.
7. Steps 3 through 6 might be repeated several times until the both parties are happy with the result and perhaps other subject matter experts will become involved in the process.
8. The technical writer finalizes the documentation and commits the changes to the repository.
9. The quality assurance team proofreads the documentation for final approval.

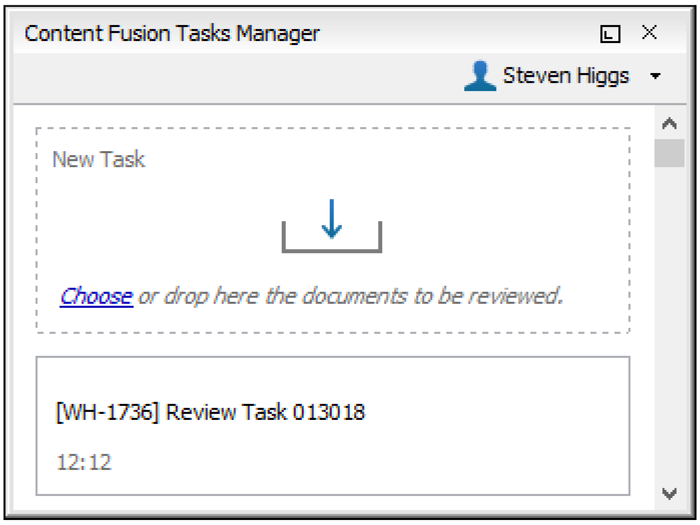
Many of our developers or other people who review the documentation do not have access to the DITA project or the same applications that the technical writer uses, thus we found that the process was often very tedious and perhaps the quality of the final documentation was compromised since the reviewers couldn't accurately visualize how the final output will look. We also found that sometimes those people involved in this process were working remotely from a different location or worked different hours, and this made the process even more tedious and slow.

So, the ultimate challenge was to create a solution that would achieve the following:

1. Provide an efficient way for documents to be passed back and forth, regardless of where the contributors are located.
2. Allow the reviewers to view and edit the documents in a simple interface that is similar to what the technical writer sees without requiring them to have access to the same projects or applications.
3. Make the process incremental and provide an intuitive way for everyone involved to communicate throughout the process.
4. Notifies the documentation team when reviewers have added messages, changed files, or finalized the review process.
5. Enable the documentation team to automatically merge the changes made by the reviewers back into their documentation project.

For this solution, we decided to develop our own new product called Oxygen Content Fusion and integrate it with our other existing products. First, we needed an efficient way for the documentation team to send the XML documents to reviewers. To achieve this, we integrated the new product with our existing Oxygen XML Editor/Author desktop application. It contributes a built-in “Tasks Manager” view where the technical writer can add documents to a task that then gets uploaded to a server. The technical writer then shares a link to that particular task with anyone they want to review the documents.

Next, we needed a simple mobile-friendly XML editing solution where the reviewers wouldn't need to have access to the documentation team's DITA project or their tools. We envisioned the reviewers only needing to have access to a browser and we wanted them to be able to open the documents in a visual editor that renders the XML structure exactly the same as the editor that the technical writer uses, and where they could proofread, add comments, and make changes directly to the source document. We already had an existing product (Oxygen XML Web Author) that is a web-based visual editor and includes features that fulfill all of those requirements, so all we had to do was integrate it into the new product. When a reviewer receives the link from the technical writer, they simply use the link to access the task in the Content Fusion interface (in any modern


Content Fusion Tasks Manager
Steven Higgs
New Task
Choose or drop here the documents to be reviewed.

[WH-1736] Review Task 013018

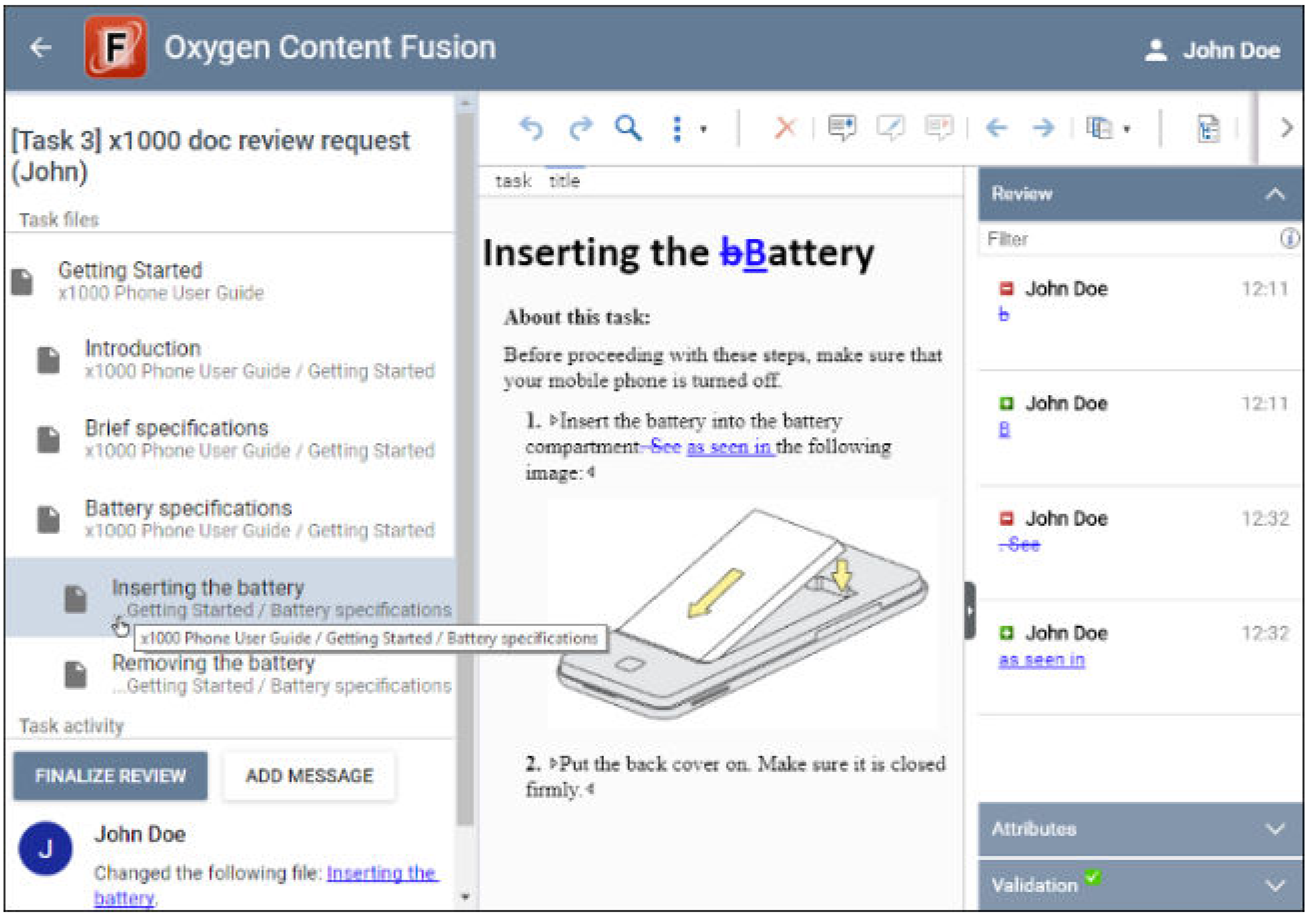
12:12

Figure 3. Content Fusion Tasks Manager

browser), and they open the uploaded files in the built-in web-based visual editor where they can make changes and add comments, We also added a default option to force the change tracking feature to always be on so that the documentation team will be able to see exactly what was changed.

Next, we wanted to provide a way for the reviewers to somehow communicate with the technical writer without having to open another application. For this part of the solution, we simply implemented a messaging mechanism within the new product. All the collaborators that have access to the task can add messages directly in the Oxygen Content Fusion interface and it was designed to have a very familiar look and feel, so it's very simple to use. Therefore, the contributors are more likely to use the feature, which often leads to better feedback and more efficient collaboration.

The next challenge was to implement a notification mechanism to inform the collaborators when a new message is added and to notify the technical writer whenever the review process progresses through the next incremental step in the workflow. The first part was achieved by simply detecting all users who had accessed the particular task at some point and sending a notification message to the email address assigned to their profile. For the second part, we decided that it was important for the technical writer to also be notified when a reviewer first accesses a task (meaning that the review process has begun), then whenever a file is changed (meaning that the review process has progressed another step), and then when a reviewer finalizes the review (meaning that the review process is finished and has progressed to a stage where the technical writer needs to act). For these notifications, we decided to not only send an email to the technical writer detected as the owner of the task but we also added a mechanism that shows a notification message in Oxygen XML Editor/Author so that the technical writer sees when the review task progresses to a new stage (or when a new mes-



[Task 3] x1000 doc review request (John)

Getting Started
x1000 Phone User Guide

Introduction
x1000 Phone User Guide / Getting Started

Brief specifications
x1000 Phone User Guide / Getting Started

Battery specifications
x1000 Phone User Guide / Getting Started

Inserting the battery
...Getting Started / Battery specifications
x1000 Phone User Guide / Getting Started / Battery specifications

Removing the battery
...Getting Started / Battery specifications

John Doe
Changed the following file: Inserting the battery.

Inserting the ~~b~~Battery

About this task:

Before proceeding with these steps, make sure that your mobile phone is turned off.

1. Insert the battery into the battery compartment. ~~See~~ as seen in the following image:

2. Put the back cover on. Make sure it is closed firmly.

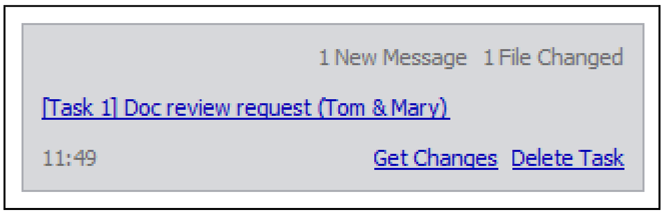
John Doe 12:11
~~b~~

John Doe 12:11
B

John Doe 12:32
~~See~~

John Doe 12:32
as seen in

Figure 4. Content Fusion Online Interface


1 New Message 1 File Changed

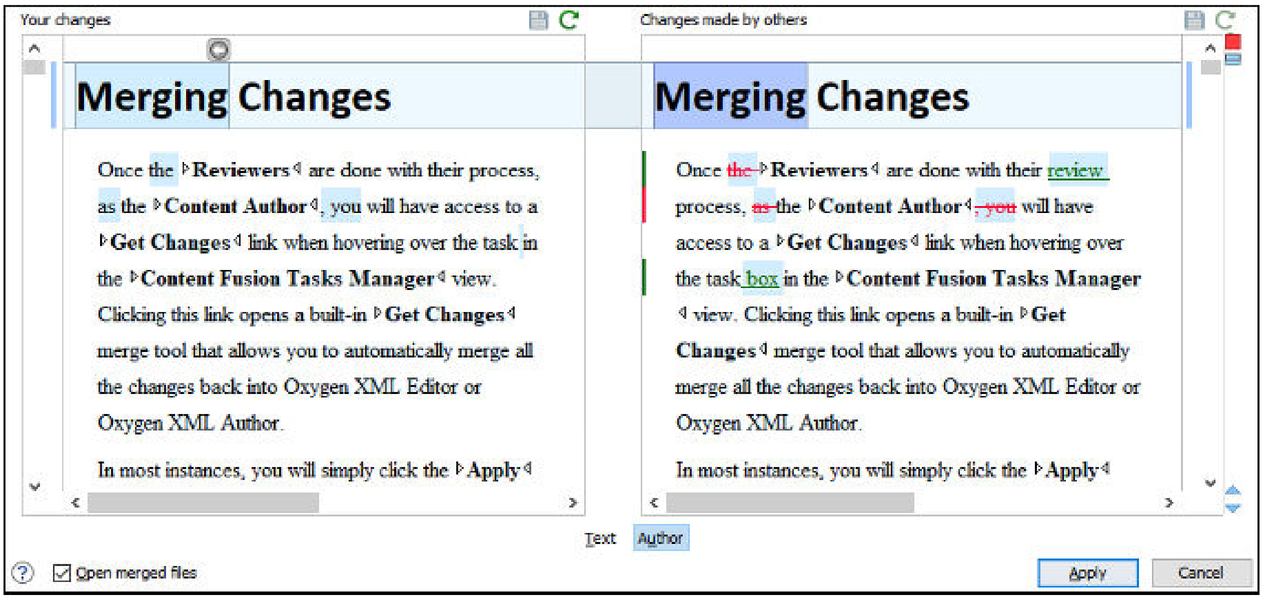
[Task 1] Doc review request (Tom & Mary)

11:49 Get Changes Delete Task

Figure 5. Task Notifications

sage is added) without having to stop their work or switching to another application.

The final challenge was to find a way for the technical writer to be able to easily and automatically merge the changes back into their project. To solve this challenge, we used our own existing file comparison tool (diff tool). We enhanced it so that it effectively looks for any changes between the original file, the changes made by the reviewer, and changes made by the technical writer after the file was uploaded. We added a visual mode so that the technical writer can view the changes in the same mode they are used to working with and in most cases, they can simply click an “Apply” button to automatically merge the changes into their project, the documents are automatically opened in the editor, and since we force the change tracking feature to always be on during the review process, the technical writer can always see exactly what was changed. In rare cases where there is a conflict, the technical writer is offered choices, such as to keep their own version of the file or overwrite it with the changes made by the reviewer.



Merging Changes

Once the ▹Reviewers◃ are done with their process, as the ▹Content Author◃, you will have access to a ▹Get Changes◃ link when hovering over the task in the ▹Content Fusion Tasks Manager◃ view. Clicking this link opens a built-in ▹Get Changes◃ merge tool that allows you to automatically merge all the changes back into Oxygen XML Editor or Oxygen XML Author.

In most instances, you will simply click the ▹Apply◃

Merging Changes

Once ~~the~~ ▹Reviewers◃ are done with their review process, ~~as~~ the ▹Content Author◃~~, you~~ will have access to a ▹Get Changes◃ link when hovering over the task box in the ▹Content Fusion Tasks Manager◃ view. Clicking this link opens a built-in ▹Get Changes◃ merge tool that allows you to automatically merge all the changes back into Oxygen XML Editor or Oxygen XML Author.

In most instances, you will simply click the ▹Apply◃

Figure 6. Merge Tool

By spending the time and resources to develop and integrate our own application to enhance our collaboration process, we have effectively addressed all of the challenges our documentation team faces when collaborating with developers or other reviewers. It's very simple and efficient for both parties. The reviewers see the documents in a visual XML editor that is very similar to what the technical writer uses, the interfaces is very intuitive, and the documents are in an advanced stage where differences in vernacular or writing styles are no longer such a challenge. The result of this has been that the documentation team tends to get better and more precise feedback from the reviewers. As for the technical writer, they can automatically merge the changes in seconds, the content always stays in the same XML form, and they always see exactly what was done. The result of this has been that the whole process is now far more efficient since numerous steps in the review process have been eliminated and it is now easier to meet deadlines since we have overcome challenges associated with collaborators being in multiple locations or having different work schedules.

4. Integrating Collaboration Tools to Streamline the Whole Documentation Process for all Departments

Our next challenge was to find a way to streamline the collaboration process between the documentation team and other internal departments, including the support team and the quality assurance department. After analyzing the workflow for those departments, we found that both the support and quality assurance personnel often reviewed topics in various forms of documentation output and then used a variety of means to share their feedback with the documentation team. In many cases, we found that this triggered a process where the communication between the two departments required multiple collaborative steps and always resulted in the documentation team making manual changes based upon various forms of feedback. Not only did we strive to make the process more consistent and streamline it by eliminating some unnecessary steps, but we also recognized that we could improve the quality and accuracy of the documentation by finding a way to allow the support or quality assurance personnel to make changes or offer feedback directly in the source document, again without requiring them to have access to the documentation team's DITA project.

For this challenge, we needed a solution that would achieve the following:

1. Provide a simple visual editor where the support or quality assurance personnel could view and edit user manual topics without requiring them to have access to the documentation team's DITA projects or applications.

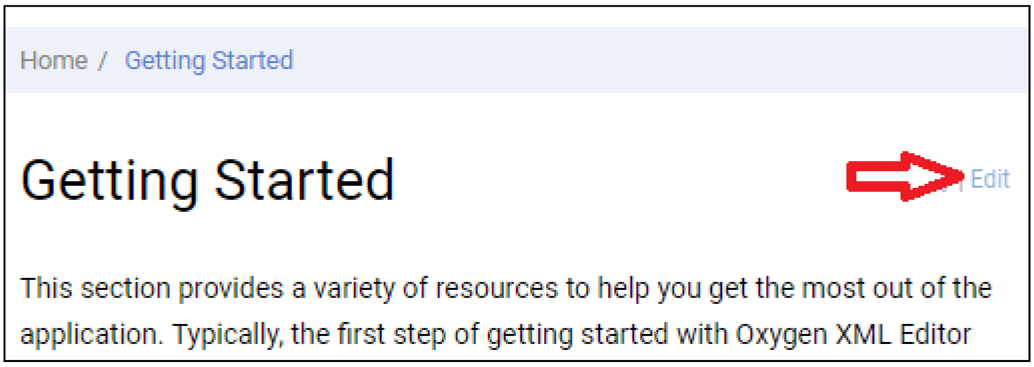
2. Integrate the solution with our Git repository so that the support or quality assurance personnel only needs to commit their changes and the documentation team would be able to easily pull those changes back into their project.

3. Make the topics accessible directly from the published output.

Again, we already had an existing product that would achieve some of the requirements. Oxygen XML Web Author already provided an online visual editor that's very similar to the Oxygen XML Editor/Author desktop product that the documentation team uses. The support or quality assurance team could use Web Author to add comments or make changes directly in the source document, and Web Author already had a mechanism in place for integrating it with a Git repository, so they already had the means to commit their changes.

We just needed a way to bring it all together so that the support or quality assurance team could easily access the particular topic without needing access to the documentation team's project. Earlier, I mentioned that we use a Web Author Review Bot to display documentation commits directly in JIRA issues and clicking a link opens the document in Oxygen Web Author. This covered cases where the collaborators needed to access the topics while progressing through stages in the JIRA issue tracking process, but we also needed a similar mechanism when the collaborators need to access a topic from published output.

To achieve this, we configured our online WebHelp and PDF output to include an “Edit” link. Clicking this link opens that specific topic in Oxygen XML Web Author where the support or quality assurance personnel can review the topic, make changes, then commit the changes to the Git repository. The documentation team is then notified of the pending changes and they merely need to pull the changes to merge them into their DITA project.



Getting Started

This section provides a variety of resources to help you get the most out of the application. Typically, the first step of getting started with Oxygen XML Editor

Figure 7. Edit Link in Published Output

The biggest advantage is that the support or quality assurance team can make corrections or add comments directly in the source topic rather than having to send their suggestions for changes in some other fashion. This helps to ensure accuracy, and thus improves the quality of our documentation and we have managed to make the process very efficient by eliminating numerous steps and making it very easy to access the content.

5. Conclusion

The number of possible solutions to refine and improve the collaboration part of the documentation process are as complex and varied as the number of possible challenges. By developing our own collaboration product (Oxygen Content Fusion), creatively configuring all the applications that we use for collaboration, integrating other existing Oxygen products (Oxygen XML Web Author, Oxygen XML WebHelp, Oxygen XML Editor/Author) with each other, other applications (JIRA), and our repositories (Git, Subversion), we have effectively addressed all the challenges that we identified in trying to make our collaboration process more efficient, effective, flexible, and fluid. Ultimately, this has resulted in effectively improving the quality of our documentation and the productivity of everyone involved in the collaboration process.

xqerl: XQuery 3.1 Implementation in Erlang

Zachary N. Dean
<contact@zadean.com>

Abstract

xqerl is an Open-source XQuery 3.1 processor written in the Erlang programming language. It compiles XQuery modules into Erlang modules that run in the Erlang virtual machine (BEAM). The goal of the xqerl project is to allow fault-tolerant, concurrent systems to be written completely – or almost completely – using the XQuery language. This paper will introduce xqerl and some of its current and future features.

Keywords: XQuery, Erlang, Concurrency, Parallel Processing

1. Introduction

An evening in March 2017 the idea of building an XQuery processor with Erlang was born. I was in between projects and out having a few drinks with friends. We chatted about XQuery and all the interesting stuff that had been added to the 3.1 version. As the night went on I began singing the praises of Erlang and how it could solve the worlds problems if it were just used more often. "Just look at WhatsApp!" Of course, this was overexaggerated, but it was getting late and seemed to make sense at least to me. "Well if it's so great why don't you make something with it?" I heard. My response, "Fine, I'll write an XQuery processor in Erlang!" The next morning, I started to work on it. Quickly I realized that this was going to take a while. I already knew of the similarities in the languages. What I later found were all the small differences that make the biggest difference. Regular expressions use a slightly different syntax, there is no concept of collations on strings, positional predicates, the list goes on. But, after ten months of working on the project full-time, it passes almost 99% of the QT3 tests for the features it implements.

2. What is Erlang/OTP?

Erlang is a general-purpose programming language and runtime environment used to build highly-available, scalable, fault-tolerant soft real-time systems. OTP (Open Telecom Platform) refers to the numerous libraries and applications that come with an Erlang distribution. It includes many middleware applications that handle distribution, an internal database, and standardized parallel process handling. Erlang was originally designed in mid-to-late 1980's by Joe Armstrong, Robert Virding, and Mike Williams at the Ericsson Computer Science Laboratory. It emerged from a project attempting to find a way to make programming telecommunication systems simpler and less error-prone. In 1998, Ericsson made the codebase for Erlang open source.

Erlang is a "concurrency oriented" functional programming language using the Actor Model. Its base concept is the use of very lightweight processes communicating with each other only by asynchronous message passing. Processes in this sense should not be confused with operating system processes or threads. The processes here are completely contained in the virtual machine and share no memory.[1]

An interesting feature of Erlang, beyond it process model, is that is can be compiled to native machine code. Since October 2001 the HiPE (High-Performance Erlang) compiler, a project originally from Uppsala University, is part of the Erlang distribution [2]. Native compilation is possible on x86 and SPARC architectures. Applications that use CPU heavily, such as xqerl, can see sizable gains in performance when compiled natively. Floating point computation can also see a slight gain when native code is used [3]. One drawback to compiling this way is the time it takes to compile code. Compilation times can be much longer when done with the HiPE compiler.

3. XQuery to Erlang pipeline - Transpilation

Compiling XQuery source code with the Erlang compiler, or more accurately transpiling XQuery to Erlang, involves many steps that are performed in a sequence. Each step can find a set of static or typing errors and stop the rest of the pipeline from being performed.

The steps are as follows:

- Scan/tokenize

  The first step in the XQuery to Erlang compilation pipeline is the tokenization of the input XQuery source code. There is a library in Erlang/OTP (leex) meant to do this for most types of source code grammars but is not currently used in xqerl for XQuery. It is however used for parsing regular expressions and JSON documents Example A.1. LEEX uses regular expressions as macros to tokenize the source document. Since XML tag names can also be XQuery keywords, this form of lexical analysis was not used. It would have caused the tokenizer to become extremely complex. Instead, a word-for-word scanner is used that maintains a state object to infer the type of token being scanned. This is inferred by location, looking back in the previously scanned tokens, and looking ahead in the source document.

- Parsing (Extended Backus-Naur Form (EBNF) notation)

Once tokenized without error, the list of tokens is processed by a parser built using the OTP library yecc. yecc is the Erlang version of Yacc (Yet Another Compiler-Compiler) used on Unix based systems. It reads a file containing the entire XQuery grammar in an Augmented Backus-Naur Form. This form is very similar to the grammar used in the XQuery specification [4]. Each match that is found in the grammar returns a tag and each of its operands. This builds the entire query into a tree that can be traversed and transformed in later stages. The following example shows two productions used in simple arithmetic statements. The symbol before the arrow is matched symbol. This is followed by the operands that have been matched out. The portion after the colon and before the dot is what will be returned, referencing the matched-out operands by position.

```
'MultiplicativeExpr' -> 'MultiplicativeExpr' '*' 'UnionExpr' :
   {multiply, '$1', '$3'}.
'AdditiveExpr' -> 'AdditiveExpr' '+' 'MultiplicativeExpr' :
   {add, '$1', '$3'}.
```

Using the example above, a simple query such as `1 + 2 * 4` will return the tree:

```
{add, 1,
   {multiply, 2, 4}
}
```

A simple but complete yecc grammar file used to parse JSON is included for reference Example A.2.

- Static analysis

  The tree returned from the parser is then recursively analyzed node for node. This is the most complex step in the transpilation pipeline.

  - Each node in tree is given a static type that is inferred by its child node types or its own type when a leaf node. Should the type not be compatible with the expected type of the statement, a type error is raised.
  - Dead branches of code, such as portions of if-then statements that can never be reached, are removed.
  - Unused global variables and functions are removed from main queries.
  - Simple operations on literal values are run in place, and their return value used instead of the original operation node. For instance, the above example returns a tree containing only the integer 9.
  - QNames are expanded based on the statically known namespaces.

  After the entire tree has been analyzed, a new, simplified tree is returned.

- Optimization phase

Currently, there is no optimization phase. This phase would be used to reorder statements to avoid multiple function calls for known values, inline positional filters or completely rewrite or simplify statements. This is planned to be implemented, at least partially, in the near future or as time* permits.

- Abstract code creation

  The statically analyzed tree is then transformed into the Abstract Erlang Format. This is a one-to-one representation of Erlang source code as an abstract tree and is the same format that the Erlang source code scanner would return had it read an Erlang source file. In this phase, the program flow is kept almost identical to the original XQuery but with one major exception. Each clause of each FLWOR statement is extracted into a local function. These functions are then either body- or tail-recursive depending on the type of clause.

- Compilation

  The final step in the pipeline is to compile the code. The binary returned from the compilation can be loaded on-the-fly into a running system, even if the same module has been previously loaded.

An example of what the output from all these steps would be if it printed Erlang source code. Comments are added for clarity:

Example 1. XQuery

```
for $x in (2,1,3)
let $y := -$x
order by $y
return $x
```

Example 2. Erlang

```
main() ->
   VarTup__1 = for__1(new), % for and let in one call
   % must return to sort entire variable tuple
   VarTup__2 = xqerl_flwor:orderbyclause(
                 VarTup__1,
                 [{fun ({_XQ__var_1, XQ__var_2}) ->
                         XQ__var_2
                   end,
                   ascending, greatest}]),
   % call return function with entire variable tuple
   return__3(VarTup__2).

for__1(new) ->
   % internally, all types are tagged
   List = [{xqAtomicValue, 'xs:integer', 2},
           {xqAtomicValue, 'xs:integer', 1},
           {xqAtomicValue, 'xs:integer', 3}],
   for__1(List);

for__1([]) -> [];
for__1([XQ__var_1 | T]) ->
   % call let__2 with head of list in parameter
   % result is head of new list, tail is call to
   % for__1 with parameter tail
   [let__2({XQ__var_1}) | for__1(T)].

let__2({XQ__var_1}) ->
   XQ__var_2 = xqerl_operators:unary_minus(XQ__var_1),
   % add XQ__var_2 to the variable tuple
   {XQ__var_1, XQ__var_2}.

return__3(List) when is_list(List) ->
   % call return__3 for every T such that T is in List
   [return__3(T) || T <- List];
% extract XQ__var_1 from the variable tuple
return__3({XQ__var_1, _XQ__var_2}) -> XQ__var_1.
```

4. Optional Features

Only some of the optional features of XQuery have been implemented thus far. This is due to the early stage of the project. The unimplemented features will be implemented as time* permits. The features that have already been implemented are the Module Feature and Higher-Order Function Feature.

Those optional features and extensions yet to be implemented are:

- Schema Aware Feature
- Typed Data Feature
- Static Typing Feature
- Serialization Feature
- Update Facility
- Full-Text

5. Data Model Conformance

As with all XQuery implementations, xqerl uses an internal XQuery and XPath Data Model (XDM). The implemented XDM imposes some limitations on the sizes and ranges of Schema Types and limits some functionality, such as a lack of the Schema Awareness feature. xqerl currently uses the XML parser (xmerl) included in the Erlang distribution. xmerl is capable of parsing well-formed XML and validating XML according to referenced DTDs.

Some limitations on ranges of data values are:

- The xs:integer type is internally represented by a bignum. Therefore, it is only limited by available memory.
- The xs:decimal type is only limited by available memory and not specific digit count. Internally it is represented by two integers, one for the entire value (all digits removing the decimal point) and the other for the position of the decimal point. When division takes place, the decimal value is rounded to 18 places to avoid overflow from repeating decimals. The limitation to 18 places is arbitrary and could be made larger.
- The types that contain a year value, xs:date, xs:dateTime, xs:gYear, and xs:gYearMonth, are limited in that the year value can have a maximum absolute value of 9999.
- xs:time and xs:dateTime types have the same limitations on their seconds values as the xs:decimal type.
- The xs:duration type and its subtypes are limited to absolute values less than 10000 years. This is to avoid durations that cannot fit into date types.
- xs:string, xs:QName, xs:anyURI and xs:NOTATION have no length limitations other than available memory. Each is internally represented by a list of codepoints.
- xs:hexBinary and xs:base64Binary types are limited to approximately 500 megabytes on 32-bit systems and 2,000 petabytes in 64-bit systems [5].

6. Moving Forward - Future work

Currently, modules in xqerl can only be called from the shell or from Erlang code. To allow xqerl to take advantage of the massively parallel and distributed capabilities native to Erlang and to make it useable in a non-development setting, further development is needed. Then next steps planned are to implement the EXPath extensions [6] and RESTful Annotations [7].

Later, compiled modules will be exposed by REST endpoints. Internally, each XQuery module or package that contains RESTXQ annotations will become a deployable Erlang application and each endpoint will become a supervisor process within the application. These supervisor processes will then in turn spawn worker processes to handle each call to the endpoints asynchronously. This loosely-coupled supervisor tree will allow for worker processes that timeout or crash and be restarted while not effecting the integrity or stability of the running system.

Another goal is to implement cost-based parallel processing within a query. This will be added during the static phase and calculate an estimated cost of complexity of each operation in the query. After a certain complexity threshold is crossed, the next higher statement in the tree that iterates a sequence will be done in parallel processes if possible.

Other features that may come later include native support for websockets, additional functions for parallel processing, and connectors for popular relational and NoSQL databases.

7. Conclusion

What started out as a "Schnapsidee", has since become a working proof-of-concept with close to 100,000 lines of code. With more work and time* it could become a feasible solution for building massively parallel XQuery solutions. So far this project has been a self-funded, one-man-show. Contributions of any kinds are always welcome that help this project reach its goal.

A. Appendix

Example A.1. leex file for tokenizing JSON - json_scanner.xrl

```
BOM        = (\x{FFFE}|\x{FEFF}|\x{0000})
S          = (\r|\n|\v|\t|\s)*
QM         = [\x{0022}]
ESCAPE     = [\x{005C}]
HEXDIG4    = ([0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f])
UNESCAPED = [\x{0020}-\x{0021}\x{0023}-\x{005B}\x{005D}-\x{10FFFF}]
ESCAPED    = ({ES-CAPE}\x{0022}|{ESCAPE}\x{005C}|{ESCAPE}\x{002F}|
              {ESCAPE}\x{0062}|{ESCAPE}\x{0066}|{ESCAPE}\x{006E}|
              {ESCAPE}\x{0072}|{ESCAPE}\x{0074}|
              {ESCAPE}\x{0075}{HEXDIG4})

UCHAR   = ({UNESCAPED})*
CHAR    = ({UNESCAPED}|{ESCAPED})*
USTRING = {QM}{UCHAR}{QM}
STRING  = {QM}{CHAR}{QM}

Rules.
{USTRING} : {token,{string ,TokenLine,TokenChars }}.
{STRING} :  {token,{esc_string ,TokenLine,TokenChars }}.

% values
false : {token,{false,TokenLine,false}}.
true  : {token,{true,TokenLine,true}}.
null  : {token,{null,TokenLine,null}}.
```

```
% number
[\-]?(0|([1-9][0-9]*))(\.[0-9]+)?([eE](\+|\-)?[0-9]+)? :
        {token,{number,TokenLine,TokenChars}}.
% structure
{S}\[{S} : {token,{array_begin ,TokenLine,array_begin }}.
{S}\]{S} : {token,{array_end   ,TokenLine,array_end   }}.
{S}\{{S} : {token,{object_begin,TokenLine,object_begin}}.
{S}\}{S} : {token,{object_end  ,TokenLine,object_end  }}.
{S}\:{S} : {token,{name_sep    ,TokenLine,name_sep    }}.
{S}\,{S} : {token,{value_sep   ,TokenLine,value_sep   }}.
{BOM}    : skip_token.
```

Example A.2. yecc file for parsing JSON - json_parser.yrl

```
Nonterminals
   root object array value values member members.
Terminals
   false true null number string esc_string array_begin array_end
   object_begin object_end name_sep value_sep.

Rootsymbol root.

root -> object     : '$1'.
root -> array      : '$1'.
root -> string     : val('$1').
root -> esc_string : str_val('$1').
root -> number     : list_to_number(val('$1')).
root -> false      : false.
root -> null       : null.
root -> true       : true.

object -> object_begin members object_end : {object, '$2'}.
object -> object_begin         object_end : {object, []}.
array  -> array_begin values array_end    : {array, '$2'}.
array  -> array_begin        array_end    : {array, []}.

members -> member value_sep members : ['$1'|'$3'].
members -> member                   : ['$1'].

member -> string     name_sep value : {val('$1'), '$3'}.
member -> esc_string name_sep value : {str_val('$1'), '$3'}.

values -> value value_sep values : ['$1'|'$3'].
values -> value                  : ['$1'].

value -> false  : false.
value -> null   : null.
value -> true   : true.
value -> object : '$1'.
value -> array  : '$1' .
value -> number : list_to_number(val('$1')).
value -> string : val('$1').
value -> esc_string : str_val('$1').

Erlang code.

val({_,_,Token}) when hd(Token) == $" ->
    lists:reverse(tl(lists:reverse(tl(Token))));
val({_,_,Token}) ->
   Token.

str_val(Val) ->
   norm_str(val(Val)).

list_to_number(List) ->
   case catch list_to_float(List) of
      {'EXIT',_} ->
         case catch float(list_to_integer(List)) of
            {'EXIT',_} ->
               #xqAtomicValue{value = V} =
                  xqerl_types:cast_as(
                     #xqAtomicValue{type = 'xs:string', value = List},
                     'xs:double'),
               V;
            F ->
               F
         end;
      Num ->
         Num
   end.

norm_str([]) -> [];
norm_str([$\\,$"|T]) -> [$" |norm_str(T)];
norm_str([$\\,$\\|T])-> [$\\|norm_str(T)];
norm_str([$\\,$/|T]) -> [$/|norm_str(T)];
norm_str([$\\,$b|T]) -> [$\b|norm_str(T)];
norm_str([$\\,$f|T]) -> [$\f|norm_str(T)];
norm_str([$\\,$n|T]) -> [$\n|norm_str(T)];
norm_str([$\\,$r|T]) -> [$\r|norm_str(T)];
norm_str([$\\,$t|T]) -> [$\t|norm_str(T)];
norm_str([$\\,$u,A,B,C,D|T]) ->
  [list_to_integer([A,B,C,D],16)|norm_str(T)];
```

```
norm_str([H|T]) ->
  [H|norm_str(T)].
```

Bibliography

[1] Getting Started with Erlang - User's Guide - Version 9.2: *Concurrent Programming* `http://erlang.org/doc/getting_started/conc_prog.html`

[2] *High-Performance Erlang Project* `https://www.it.uu.se/research/group/hipe/`

[3] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, T. Lindahl *All you wanted to know about the HiPE compiler* (but might have been afraid to ask) `http://erlang.org/workshop/2003/paper/p36-sagonas.pdf`

[4] *XQuery 3.1 Grammar* EBNF `https://www.w3.org/TR/xquery-31/#nt-bnf`

[5] *Efficiency Guide - User's Guide - Version 9.2* Advanced `http://erlang.org/doc/efficiency_guide/advanced.html`

[6] *EXPath - Modules* `http://expath.org/modules`

[7] *RESTXQ 1.0: RESTful Annotations for XQuery* `http://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html`

XML Tree Models for Efficient Copy Operations

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

A large class of XML transformations involves making fairly small changes to a document. The functional nature of the XSLT and XQuery languages mean that data structures must be immutable, so these operations generally involve physically copying the whole document, including the parts that are unchanged, which is expensive in time and memory. Although efficient techniques are well known for avoiding these overheads with data structures such as maps, these techniques are difficult to apply to the XDM data model because of two closely-related features of that model: it exposes node identity (so a copy of a node is distinguishable from the original), and it allows navigation upwards in the tree (towards the root) as well as downwards. This paper proposes mechanisms to circumvent these difficulties.

1. Introduction

An XSL transform takes linear time
If the input and output are almost the same.
Though the changes you make may be local and small
You still pay the price of transforming it all.

Many XML transformations, whether expressed in XSLT or XQuery, copy large chunks of the input directly to the output, without change. This is typically an expensive operation, requiring both time and memory proportional to the size of the subtree being copied. This cost is particularly painful when a transformation operates incrementally, making many passes over the input, each of which only makes small changes.

For an example of such a problem, see [2]. In that paper I explored the possibility of writing an XSLT optimizer in XSLT. Optimization is essentially a series of transformations applied to an expression tree, so it is in principle a task to which XSLT should be well-suited; but my conclusion in that paper was that it wasn't feasible to achieve adequate performance, largely because each transformation step involved copying the large parts of the tree that remained unchanged. Recently in Saxonica we have been revisiting this problem (see [4]) because we are interested in making the entire XSLT compiler portable across platforms, and the classic way of achieving this is to write the compiler in its own language. This led me to look again at the efficiency of transformations that make small changes to a large tree.

The fact that copying a subtree is expensive is a consequence of two particular rules in the XDM data model: (a) nodes have identity (which means that the expression `copy-of(X) is X` must return false – a copy of a node is not the same thing as the original), and (b) nodes have parents (which means that `exists(copy-of(X)/..)` must return false – when a node is copied, the copy is parentless). Any implementation of a copy operation that retains these properties without performing a physical copy of the subtree is going to be complicated.

In the XDM data model, the maps and arrays used to represent JSON structures do not have this property. In the tree representation of JSON, there is no way of navigating from an object to its parent; and there is no way of distinguishing two copies of the same object. This means that subtrees can be shared, which makes copying logically unnecessary (or to put it another way, producing a logical copy does not require producing a physical copy).

Saxon[1] implements maps and arrays using what I will call "versioned" data structures. (The names "immutable" and "persistent" are also used, but both adjectives have alternative meanings, so I will avoid them). In a versioned data structure, after any update operation, both the old and the new values are available for further processing, yet the new value shares memory with the old for those parts of the data that have not changed. Appendix A describes briefly how versioned maps and arrays work. A versioned data structure for XML trees is more difficult to achieve, because of the problems of node identity and parent navigation.

There's no intrinsic difference between XML and JSON at the lexical level that accounts for this deep difference between the way that XDM models the two cases. We could explore what happens when we add parent pointers to JSON trees, or we could explore what happens when we remove them from XML trees. This paper does the latter.

The ability to navigate from a node to a parent (and therefore, implicitly, to its siblings) is extremely useful, because it makes it possible to identify nodes of interest by their context as well as their content. In [3] I showed some use cases where XSLT 3.0 is applied to the task of transforming JSON, and the inability to access this contextual information proved a constant obstacle, to the extent that I concluded the easiest way to accomplish many JSON transformations was to convert the data to XML, transform the XML, and then convert it back to JSON.

[1]Some statements made here about the Saxon product refer to code that is implemented and tested but not yet released.

Now the freedom to navigate upwards and down,
to parents, descendants, children and peers,
Means the rule for transforming a node in your text
Can refer to the context in which it appears.
So if `xml:lang` *says your paragraph's Dutch,*
This may affect formats for numbers and such,
But to determine what language applies at each point
You must know the container in which it is found.

In the latest incarnation of the Saxon-JS product, we are using a JSON-based model internally to represent the interpreted expression tree, but we have found it necessary to introduce parent pointers to allow access to context information such as the static base URI and in-scope namespaces of an expression. This doesn't cause any problems in this case because the expression tree, by the time the compiler is finished with it, never changes.

Now: the main thrust of this paper is to show that providing the ability to navigate from a node to its parent does not necessarily imply that the stored tree needs to include parent pointers. There's another way to enable access to the parent, which is to remember, when you get to a node, how you got there. There's no way of getting to a node without going via its parent, so in principle you can always retrace your steps. Knowledge of the parent can thus be part of the information returned when a node is retrieved, even if the information is not actually stored with the node. Equally, the identity of a node (affecting the result of the XPath `is` operator) can be a function of how the node was reached: if we treat the identity of a node as a list comprising the identities of all its ancestors, that is, a path to the node from the root, then it does not matter if a physical subtree is shared by several logically separate XML documents: a node can be reached by more than one path, but it is the full path that establishes the node identity, so such a node has multiple identities depending on how you got there. With this insight, we can see that it should be possible to provide full XPath navigation capability on a tree with no stored parent pointers and no built-in notion of node identity.

The KL-tree described in section 3 is an implementation of this concept.

2. Push and Pull Processing

The semantics of XSLT 1.0 were written in terms of instructions such as `xsl:element` *writing nodes to the result tree*: the narrative was written assuming a push model where instructions push data to a destination. By contrast, for XQuery and later XSLT versions, the specification uses *pull* language: element constructors are expressions that return a result to their caller, namely a newly constructed element. The pull model implies bottom-up tree construction, where leaf nodes are constructed first, and then grafted onto their new parents: this inevitably involves copying the node to give it a new parent, at each level of construction.

A processor that implements this literally as written is going to be very inefficient, because of the amount of tree copying needed. In practice there are a number of ways the repeated copying can be avoided:

- The implementation can use a push model internally. What happens here is that an instruction like `xsl:element` starts by emitting a *startElement* event to an output receiver; it then processes its child instructions (also in push mode), and finally emits a corresponding *endElement* event. If the output receiver is a serializer writing lexical XML, this approach means that the result nodes are never actually constructed in memory at all: the processor emits a sequence of events which are translated into lexical XML markup by the serializer. If the output is an intermediate tree, the receiver will be a tree builder that uses the stream of incoming events to construct an in-memory tree, but none of the intermediate nodes will ever need to be copied. This is the approach used by the Saxon-Java product.
- The implementation can use a bottom up model, and attempt to recognize where leaf nodes do not need to be copied, but can instead be directly grafted to their new parent by updating a parent pointer. This relies on being able to recognize that the leaf node exists solely for the purpose of creating the content of the next container, and will never be used as a parentless node in its own right. This isn't quite as effective as the push strategy, because it involves materializing the result tree prior to serialization, but it can still perform well. This is the approach used in Saxon-JS.

This paper suggests that a third approach might be possible, which is simply to ensure that copying a tree of nodes is extremely fast: ideally it should cost nothing. This is achieved by creating a virtual copy of the tree: a separate tree in terms of XDM node identity, but sharing the same underlying storage as the original. Our first attempt at this is the KL-tree, described in the next section.

3. The KL-Tree

This section describes the KL-Tree, an experimental implementation of the XML part of the XDM data model.

The data that physically exists in memory is the K-Tree, and its nodes are called K-nodes. Putting attributes and namespaces to one side for the moment, we have five node kinds: documents, elements, text nodes, comments, and processing instructions. Since comments and processing instructions behave just like text nodes, we can ignore them for the purpose of this discussion.

So, as a first approximation, the K-tree contains:

- Document nodes, which contain a sequence of child nodes
- Element nodes, which have a name, a sequence of child nodes, plus attributes and namespaces
- Text nodes, which contain a string value

K-nodes do not contain enough information to enable navigation to ancestors or siblings, or to enable sorting of nodes into document order. To achieve that, any navigation through the K-tree returns not the K-node itself, but an L-node; an L-node contains a reference to the K-node, plus additional information. Specifically, the additional information in an L-node comprises a reference to its parent L-node (with null used to indicate that the L-node is the root of the L-tree), plus the position of the L-node among its siblings in the L-tree. With this additional information, navigation from an L-node using any of the 13 XPath axes becomes possible, as does sorting into document order.

Our initial invention to answer this question
Was a tree in which nodes pointed down but not up.
Elements reference children and text nodes;
The link is one way: you can only descend.
But now when a query selects a descendant,
We remember the path for retracing our steps.
The pointer to parent becomes now redundant
We can find a container, whatever our depth.

The L-nodes are created on demand, when a node is retrieved in the course of navigation, and they are garbage-collected as soon as they are no longer needed. With a little bit of optimization, it is possible in many cases to avoid creating L-nodes that aren't needed, for example with an XPath expression `child::title`, we can arrange only to create L-nodes for those K-nodes that match the required name.

Two L-nodes are identical (in the sense of the XPath `is` operator) if their parents are identical and they have the same sibling position; so it's only root nodes that have intrinsic identity. It doesn't matter whether the two L-nodes are represented by the same Java object, or whether the K-nodes that they reference are represented by the same Java object: one Java object can represent several nodes, and several Java objects can represent the same node.

Similarly, sorting of L-nodes into document order can be achieved from knowledge of the parent nodes and sibling positions.

4. KL-tree Performance

Appendix A summarizes the execution time of various important operations, comparing the KL-tree implementation with Saxon's standard TinyTree implementation as well as the more conventional LinkedTree model, and (for completeness) the other tree implementation models supported by Saxon.

What these figures show is that the KL-tree is dramatically faster for one particular operation, that of grafting a tree into a new containing tree, but it is a little bit slower than the existing TinyTree implementation for many other operations. In particular, searching the KL tree is about 4 times slower. The KL-tree also uses more memory. This is not because the model is intrinsically inefficient; it just fails to reproduce some of the optimizations implemented in the TinyTree. The TinyTree achieves much of its fast search time by using arrays of data rather than linked objects to represent nodes, and because scanning an array is faster than following pointers in a linked list, it is hard for any implementation using linked lists to achieve comparable performance.

Sadly, this appears to be a show-stopper as far as incorporation into Saxon is concerned. The number of stylesheets that show an overall performance improvement from the KL-tree is small, and moreover, it's difficult to recognize them by static analysis. This means that the feature is only viable as a user-selected option, and we know from experience that only a very small number of users who stand to benefit from tweaking such features will actually understand the feature sufficiently well to take advantage of it. If only 5% of stylesheets stand to gain, and if only 5% of the authors of those stylesheets recognize the fact, then adding the feature will not create enough happiness in the user community to make it worth the trouble.

So let's throw this idea out of the window for the time being (Prague being a popular place for defenestration) and try something else. Since the TinyTree is delivering good all-round performance, let's see if we can use that as our baseline, and make incremental improvements.

5. The TinyTree

At this stage we need to explain the workings of the TinyTree, which is Saxon's default tree implementation. Although the data structure has been around for many years, and has changed very little, the only published information is the low-level internal Javadoc `https://www.saxonica.com/documentation/index.html#!javadoc/net.sf.saxon.tree.tiny/TinyTree`, plus a slightly out-of-date blog article [1].

The data structure consists of a set of arrays, held in the TinyTree object. The arrays are in three groups, where in each group the arrays can be considered to represent columns in a table. Using Java arrays to represent the columns of the table, rather than the conventional approach of using one Java object per row, accounts for much of the space saving benefits, and also provides for fast tree construction and navigation.

The TinyTree structure makes no use of pointers;
Its content instead is arranged using vectors.
One holds the depth, a second the node kind,
A third holds the names, coded as numbers.
A search for descendants will step through these vectors
Comparing the node kind and name for a match.
With no pointer chasing, and no string comparing,
The search for a node is impressive to watch.

The principal table contains one row for each node other than namespace and attribute nodes. These rows are in document order. The following information is maintained for each node:

- the depth in the tree
- the name code
- the index of the next sibling
- two fields labelled *alpha* and *beta*, described below
- the type annotation that results from schema validation (this array is absent for untyped trees)
- the index of the preceding sibling. This array is created lazily only when needed, the first time that the preceding-sibling axis is used for any node in this tree.

The meaning of *alpha* and *beta* depends on the node kind. For text nodes, comment nodes, and processing instructions these fields index into a string buffer holding the text. But for element nodes, *alpha* is an index into the attributes table, and *beta* is an offset into the namespaces table. Either of these may be set to -1 if there are no attributes or namespaces.

A name code is an integer value that indexes into the *NamePool* object: it can be used to determine the prefix, local name, or namespace URI of an element or attribute name. Name codes enable searching for elements and attributes using fast integer comparisons rather than string comparisons.

The attribute table holds the following information for each attribute node:

- a pointer to the attribute's parent element
- prefix
- name code
- attribute type
- attribute value

Attributes for the same element are adjacent.

The namespace table holds one entry per namespace declaration or undeclaration (*not* one per namespace node). The following information is held:

- a pointer to the element on which the namespace was declared or undeclared
- namespace prefix
- namespace URI

The links between elements and attributes/namespaces are all held as integer offsets. This reduces size, and also makes the whole structure relocatable. All navigation is done by serial traversal of the arrays, using the node depth as a guide.

Saxon attempts to remember the parent of the current node while navigating down the tree, and where this is not possible it locates the parent by searching through the following siblings; the last sibling points back to the parent. In the case where there is a large number of siblings, occasional parent pointers are inserted as pseudo-nodes to reduce the length of this search.

6. Virtual Copy

Existing Saxon releases include an optimization whereby an expression of the form

```
<xsl:variable name="x">
    <xsl:copy-of select="$doc//a/b/c"/>
</xsl:variable>
```

creates a virtual copy of the selected `<c>` element nodes, rather than doing a physical copy.

Rather like an L-node in the KL-tree model, the virtual copy is a wrapper node that points to the original node of which it is a copy. Many of the properties of the virtual node (for example, the name, type, and string value) are identical to the corresponding properties of the original. The mechanism can also handle some variation, for example there is scope for the original data to be schema-typed, while the copy is untyped. Navigating around the virtual tree is done, by and large, by navigating around the underlying physical tree, and then wrapping the resulting node. The main way in which the virtual copy differs from the original (apart from having a different identity) is that XPath navigation never strays outside the subtree that has been copied. Navigation from any node in the tree to its ancestors stops when it hits the root of the virtual copy; navigation from the root to siblings or parent returns an empty sequence.

Unlike the KL-tree, the virtual copy cannot be shared as a child of multiple parents. In fact, a virtual copy is always a parentless copy of some original tree or subtree: the original node may or may not have a parent, but the virtual copy never has. This gives it limited usefulness. Indeed, one could argue that it is only ever used to ameliorate code that was badly written in the first place, because it is essentially used only to eliminate copying that was never necessary. The relevant variable could equally well have been written as:

```
<xsl:variable name="x" select="$doc//a/b/c"/>
```

with no copying needed.

Although this mechanism has limited usefulness in its current form, it turns out not to be difficult to extend it. In particular, we can extend it so that:

- A virtual copy *V* is identified by a pair of nodes (*R*, *P*), typically in different trees. *P* is referred to as the grafting host: we are effectively grafting the tree rooted at *R* to a new parent *P*.
- *V* is deep-equal to *R*: they have isomorphic subtrees that share the same storage
- The parent of *V* is *P*, which in general is not the parent of *R*.

When navigating *V*, the result of any navigation within the subtree is a wrapper node (like the L-node described earlier) which remembers that the parent of *V* is *P* rather than *R*; and any navigation that strays from the subtree (which in practice will always reduce to a call on `V.getParent()`) uses this information to return to the tree containing the grafting host.

We can now look at extending the TinyTree model so that an element node in the model (which in the normal way would be immediately followed by all its descendant nodes in document order) can be replaced by a reference to an external element node which is deemed to be copied at the relevant position.

So the tree representation produced by the following construct:

```
<xsl:variable name="x">
    <out>
        <xsl:copy-of select="$doc//a/b/c"/>
    </out>
</xsl:variable>
```

would contain an entry for the document node at level 0, then an entry for the wrapper `<out>` element at level 1, then a number of "external element" nodes at level 2, each containing some kind of reference to one of the selected `<c>` nodes. For convenience, we'll call this the "host tree", and we'll refer to the trees containing the `<c>` nodes as "grafted trees". Of course, the process can be nested arbitrarily deep, in that grafted trees can themselves contain external element nodes to further copied trees.

How do we navigate such a structure?

Firstly, if we are processing the descendant nodes in the outer tree in document order, then when we hit an external element node, we have to remember where we are on some stack, and continue by processing the descendants of the grafted node. When we've finished scanning the grafted subtree, we pop the stack. This part isn't too difficult.

More tricky is that when we're processing the grafted tree, we have to remember where we came from. The rules are similar to those for the current VirtualCopy described in the previous section, but with some key differences:

- The parent of the root node in the grafted tree is no longer absent, it is the parent of the external element node in the outer tree.
- Similarly, the siblings of the root node in the grafted tree are the siblings of the external element node.

With these changes, a `copy-of()` operation on a TinyTree node creates a parentless VirtualCopy (as today), and an operation that attaches such a VirtualNode to a new parent, in the course of building a new TinyTree, adds a reference to the VirtualCopy. Both cases now take constant time independent of the tree size.

A minor refinement: for very small trees, for example those consisting of a single element node with a single text node child, it may turn out to be cheaper to perform a physical copy of the node.

Unfortunately, though, most of the implicit copying that happens during a transformation isn't done with explicit `copy-of()` operations, it is done using the recursive shallow copy implicit in the built-in template rules. So the next section studies how we can make recursive shallow copy equally efficient.

7. Shallow Copy and the Identity Template Pattern

Returning to the original use case, stylesheets that make small changes to large trees are often written to use a design pattern with a fallback rule that shallow-copies an element, overridden by higher-priority rules to make specific changes. For example, a stylesheet to delete all `<Note>` elements might be written like this in XSLT 3.0:

```
<xsl:mode on-no-match="shallow-copy"/>
<xsl:template match="Note"/>
```

In earlier XSLT releases the default action would be spelled out explicitly, for example:

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

The problem here is that the code is copying elements from the source tree to the result tree one node at a time, which makes it difficult to take advantage of a fast deep copy.

The first thing we have to do is to detect that this pattern is in use: this is clearly easier to do with the declarative XSLT 3.0 approach. It's harder when the identity template is written out explicitly, because there are many variations on how it is written; however it should be possible to detect the common cases.

When we detect that the template rules for a mode use shallow-copy as the fallback action, with specific actions for a small number of match patterns, we can attempt an optimization: if there is no explicit template rule for an element, or for any of its descendants (or for their attributes, if applicable) then the element can be deep-copied to the result, with no further processing of the subtree.

The identity pattern in XSLT
is troublesome mainly because you can't see
which nodes have subtrees that don't change one jot
and would benefit greatly from copying the lot.

If the stylesheet is schema-aware then there is potential to recognize statically that there are some elements that will always be deep copied. Sadly, however, writing schema-aware stylesheets seems to have remained a minority interest, so this approach on its own won't get us very far. However, there might still be benefits from doing a dynamic check.

Specifically, if the following situation arises dynamically:

- `xsl:apply-templates` selects a node *N* for which there is no matching template rule
- the current mode (explicitly or implicitly) uses `on-no-match="shallow-copy"`
- *N* is a node in a TinyTree
- the instruction is evaluated in push mode
- the current output destination is a TinyTree builder

then it may be worth scanning the descendants of N to see whether any of them matches an explicit template rule; and if not, performing a deep copy-of() operation rather than a recursive tree walk.

The approach could be further improved with a learning strategy: if (say) the first ten nodes with a particular name M have been found to have descendants matched by an explicit template rule, then it's probably not worth considering this approach for any subsequent nodes named M.

An important caveat is that this tactic will never be used in the common case where transformation results are being written directly to a serializer. The cost of producing serialized XML will always increase with document size. The benefit comes when a pipeline of transformation phases pass data to each other in the form of in-memory trees (and it applies whether these transformation phases are written as a sequence of separate XSLT stylesheet executions or as a single XSLT execution).

8. Use in XQuery Update

In XQuery, the code for copying a tree with minor changes to selected nodes is somewhat tedious to write: essentially, the `xsl:apply-templates` mechanism needs to be simulated with a recursive function that switches on the type of the supplied node using a `typeswitch` expression, each branch typically containing a recursive call to process child nodes using the same logic. In principle, the same optimizations could be applied as for the XSLT shallow copy pattern; but it is probably harder to detect this pattern in XQuery because there is more scope for minor variation in the code.

A second way to make small changes to a document in XQuery is to use XQuery Update. For example a query to delete all the `Note` elements at any level could be simply written:

```
delete node //Note
```

The downside of this mechanism is that the updated document (with the `Note` elements deleted) is not visible within the same query. Instead, some external mechanism (perhaps XProc) is needed to insert the updating query into a pipeline of operations. The XQuery update specification states that modifying nodes within a tree does not affect the node identity of other nodes: in effect, the tree becomes mutable; except that there is no way within the XQuery language of comparing the node identity before and after update to see whether the claim is actually true.

If updates are to be made visible within a query, the only way to achieve this is with the "copy-modify" expression (also known as a transform expression). An example might be:

```
copy $doc2 := $doc
modify (delete node $doc//Note)
return $doc2
```

In the sadly-abandoned 3.0 version of XQuery Update, this can be replaced with the simpler syntax:

```
$doc transform with {
   delete node .//Note
}
```

The result of this expression is now a copy of the subtree rooted at `$doc` in which the `Note` elements have been deleted. Pure functional behaviour and immutability have been restored by constraining the in-situ modification to work on a tree that is created as a copy of the original, where the copy becomes accessible to the rest of the query only in its modified form.

So we are once again in the situation where the cost of making this change depends on the size of the document, and not only on the number of nodes being changed.

With a construct like this, however, we have a very much better chance of exploiting a virtual copy mechanism. The first stage in evaluating this copy-modify expression is to produce a pending update list, which is a list of actions to be applied to the tree, together with the nodes that they affect. We can then expand this list to include all the ancestors of affected nodes. Then, when performing the copy operation to construct `$doc2`, we can implement this by means of a recursive copy on all its children, and in the course of this recursive deep copy, any node that is `not` on the expanded list of affected nodes can be virtually-copied by creating a reference to the original node in `$doc`.

In XQuery Update it's easy to see
that the modified nodes form just part of the tree.
Thus the list of those nodes that remain unaffected
Is readily formed, as might be expected.
And with virtual copies of parts that stay constant
Applying small changes takes only an instant.

This promises to be sufficiently useful that we could consider providing XSLT syntax with the same semantics: the above example might become:

```
<uxsl:update select="$doc">
  <uxsl:delete select=".//Note"/>
</uxsl:update>
```

while a more complex example might be:

```
<uxsl:update select="$doc">
  <uxsl:delete select=".//Note"/>
  <uxsl:rename select=".//Comment" to="Remark"/>
  <uxsl:replace-value select=".//Salary" by=". * 1.1"/>
</uxsl:update>
```

The result of the `uxsl:update` expression would be the updated copy of `$doc`

References

[1] Michael Kay *Saxon: Anatomy of an XSLT Processor*, 2005. Article published at IBM DeveloperWorks. Available at `https://www.ibm.com/developerworks/library/x-xslt2/`

[2] Michael Kay: *Writing an XSLT Optimizer in XSLT.* Presented at Extreme Markup Languages: Montréal, Canada, 2007. Available at `http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html` and at `http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html`

[3] Michael Kay: *Transforming JSON using XSLT 3.0.* Presented at XML Prague, 2016. Available at `http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf` and at `http://www.saxonica.com/papers/xmlprague-2016mhk.pdf`

[4] John Lumley, Debbie Lockett and Michael Kay: *Compiling XSLT3, in the browser, in itself*. Presented at Balisage: The Markup Conference 2017, Washington, DC, August 1-4, 2017. In Proceedings of Balisage: The Markup Conference 2017. Balisage Series on Markup Technologies, vol. 19 (2017). Available at `https://doi.org/10.4242/BalisageVol19.Lumley01`

[5] `http://www.xml-benchmark.org`

A. Measurements

This appendix gives measured timings (in milliseconds) for various operations on various implementations of XDM trees. The measurements were made on an experimental version of the Saxon XSLT processor; the precise measurement configuration is not significant because we are only interested in the relative numbers. The data used was a 10Mb version of the XMark dataset[5].

The operations that we measured are as follows:

- Build: the time taken to build the tree by parsing a 10Mb source document.
- Scan: time taken to scan the tree in descendant order, sending SAX-like events to a sink receiver that immediately discards the data.
- Graft: time taken to build a new tree consisting of a document node, a wrapping element node, and a copy of the 10Mb source tree.
- Scan Grafted: time taken to scan the tree that results from the graft operation, again sending SAX-like events to a sink receiver that immediately discards the data.
- Search Grafted: time to execute the XPath query `count(//*[@id])` (which returns 6075 elements) on the tree that results from the graft operation.

These operations were timed with the following implementations of the XDM tree model:

- TinyTree (old): the TinyTree as available in the released Saxon product, without special support for virtual copying
- TinyTree (new): the TinyTree modified as described in this paper, to allow grafting of a virtual copy of a subtree
- Linked Tree: the Saxon Linked Tree, a conventional tree implementation where nodes are Java objects with pointers to children and parent nodes
- KL-Tree: the experimental KL-Tree model described in this paper
- DOM: the implementation of the W3C DOM interface packaged in the Oracle JDK
- Domino: a hybrid tree implementation introduced in Saxon 9.8, consisting of a DOM supported by additional indexes for fast searching

- XOM: see `https://xom.nu`
- JDOM2: see `https://www.jdom.org`
- DOM4J: see `https://dom4j.github.io`
- AXIOM: see `https://ws.apache.org/axiom/`

Here are the measurements:

Table A.1. Measurements of various operations on various tree implementations

Model	Build	Scan	Graft	Scan Grafted	Search Grafted
TinyTree (old)	120	6.6	85	9.2	6.3
TinyTree (new)	120	7.2	0.8	6.8	7.4
LinkedTree	125	19	65	25	15
KL-Tree	118	20	0.005	20	58
DOM	133	127	216	120	37
Domino	211	51	114	56	14
XOM	184	55	197	90	42
JDOM2	133	64	164	86	30
DOM4J	134	96	188	108	205
AXIOM	145	82	158	98	127

How it was measured: using a microbenchmarking environment in Java, calling low-level interfaces to simulate the run-time activity of an XSLT or XQuery processor. Each test was run repeatedly for 10 seconds or more to warm up the Java hotspot compiler; this was then repeated for another 10 seconds to get an average execution time, and only the figures for the second run were recorded.

B. Versioned Maps and Arrays

Given a map such as

```
let $old := map{"a":1, "b":2, "c":3}
```

it is possible in XPath 3.1 to perform an operation such as

```
let $new := map:put($old, "b", 17)
```

whose result is a new map,

```
map{"a":1, "b":17, "c":3}
```

After this operation, $old still refers to the original map, while $new refers to the new map. But the map:put() operation does not copy parts of the map that have not been changed: the cost of the map:put() operation is essentially independent of the size of the map that is being modified.

Various data structures can be used to achieve this effect. The one that Saxon uses is a hash trie.

To simplify the actual implementation, we can consider that for each possible key value there is a hash code which can be viewed as a sequence of seven 5-bit integers. A tree of depth 7 with a fan-out of 32 can then be used to locate any value: an entry in the leaf nodes of this tree is a list of key-value pairs, where the keys are those sharing the same hash code.

Modifying the entry for one particular key then involves replacing 7 nodes in the hash trie, one for each level corresponding to the 7 components of the hash code. This will always involve a replacement for the root of the tree. All nodes in the tree other than these 7 can be shared between the old tree and the new. Modification thus has a constant cost: whether the map contains one entry or a billion, the put() operation creates exactly 7 new nodes in the tree.

In practice the actual hash trie implementation has optimizations that reduce the cost of handling very small maps, because these are quite common. For example a map of less than five entries is implemented as a simple list of key-value pairs.

A similar solution is used for arrays. In concept, an array is simply implemented as a map whose keys are integers. But because the structure needs to handle operations other than get() and put() efficiently (notably, retrieval of entries in key order), and because integers are not limited to 35 bits, the actual trie structure used internally is different.

See also: Wikipedia, *Persistent Data Structures*, `https://en.wikipedia.org/wiki/Persistent_data_structure`

Using Maven with XML development projects

Christophe Marchand
Oxiane
<cmarchand@oxiane.com>

Matthieu Ricaud-Dussarget
Lefebvre Sarrut
<m.ricaud-dussarget@lefebvre-sarrut.eu>

1. Context

I'm a Java developper, since many years. I've seen and used many build systems, from a simple document that explains how to build, to a configured build descriptor. Most of build systems I've used are script-like systems. Maven is a build descriptor, where all the build phases are configured, not scripted. Maven has been used since 2009, and widely used since 2012. I've worked on many projects where provided data was XML. Those projects where mostly Java projects, embedding few XML technologies, as XPath and XSL. In 2015, I started a new contract in ELS, a publishing company, where the most important part of code were XML languages, as XSL, XQuery, XProc, RelaxNG, and so on ; they are all familiars to you.

I've been very surprised that some projects didn't used correctly Source Control Management, that some projects where deployed on servers from a SVN checkout, that some projects did not have unit tests, that there were no standard way to build a project, and to deploy it on a target box.

I've started to work to define a standard way to define a project, to organize sources, to build, to run unit tests, and to define a way to avoid code duplication.

2. Needs

We had many requirements on the development organization:

- we must ensure that code is not duplicated anywhere in our projects
  - Maintaining such code properly becomes a nightmare as the time goes
  - XML technologies generally have a strong ability to deal with overriding rule (xsd, xslt, etc.): it makes possible to create common code at any level in a logical architecture
  - It helps in creating specific/generic code architecture

    - It improves quality ; *That implies a simple way to re-use existing code*
- we want common code changes won't break every projects using it:
    - we want to be able to separate each chunk of code, and identify each version with no ambiquity *to produce deliveries that can be identifed as deliveries and not as source code*. Let's call this an *artifact*
    - we must be able to distinguish a stable release (an artifact we know exactly which commit of code has produced it), from a development one (an artifact that is still under development, and that may change from one day to another)
    - we must ensure that a release artifact can not be re-build: i.e. a release artifact can not be modified
    - when we re-use a piece of existing code, we want to reference it, through a release artifact reference ; hence, we are sure the referenced code will never be modified we want every artifact version being accessible from any other project easily we need to publish build artifacts to a central repository ; this central repository will be then used to get artifacts when needed
- we must ensure that unit tests are always successfully run before building an artifact
- we must be able to deploy programs to target locations from the central repository
    - a "program" is usually an aggregate of many artifacts, with a shell
- last but not least, we want all these requirements to work the same way for all our editorial languages: XSLT, XQUERY, SCHEMATRON, DTD, XSD Schema, Relax NG, … and Java ! All programs we deploy are Java programs, either run directly, or launch by an orchestrator. Finally, we deploy Java programs, who run XSLT, XQuery, or other XML languages.

We also had some wishes:

- we'd like that release artifacts can only be build by a dedicated build environment ; this ensures that build command and options are always the same, and that build is not performed locally by a developer, with special options. Well, we'd like that release buid could be repeat in case of a massive crash.
- we'd like to deploy only compiled code
    - most of language specifications define a compile process (well, static errors, at least)
    - only XSLT 3.0 has defined that a XSL can be compiled, moved, and then run elsewhere. So, a compiled form exists, even if it is not standardized.
    - other languages may defined some compile-like process,

    - we may have some transformations to apply to source code before it can be accepted as "compiled"
    - we could define that compilation step is any operation that transform a source code to a build code.
- we'd like to be able to generate code
- we'd like to be able to validate an XML file, as a condition to build artifact
- we'd like that developer documentation will be published on a Web server each time a build is performed. Hence, a developer who wants to use a particular artifact is able to find the documentation of this artifact.

3. Solutions

ELS has tested various tools and frameworks to manage their project management requirements (mainly XProject, EXPath, ant). XProject defines a project structure, but lacks on version management. EXPath has a repository, but each repository should be manually fed by modules ; it is widely used by XSLT and XQuery engines, but not really suited to create Java programs that embed EXPath modules, and is so not suited to a scalable architecture, where Java components could be dynamically deployed on many servers. Ant allows everything, but ant is a scripting system, and scripts should have their own unit tests, which is never done.

The only one that satifies all requirements is Maven. But Maven does not provides plugins for a lot of tasks we need.

Maven has a standard way to build: phases lifecycle. Build has a lifecycle, and phases are sequentially organized through this lifecycle ; one phase can not be executed if all previous phases have not been successfully executed.

According to Maven documentation[1], lifecycle phases are:

- validate
- initialize
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile
- process-classes
- generate-test-sources
- process-test-sources
- generate-test-resources
- process-test-resources

[1]http://maven.apache.org/ref/3.5.0/maven-core/lifecycles.html

- test-compile
- process-test-classes
- test
- prepare-package
- package
- pre-integration-test
- integration-test
- post-integration-test
- verify
- install
- deploy

Some phases are not in lifecycle, and do not have prerequisites:

- clean
- site

At each phase, plugins are bound. When a phase is executed, all plugins bound to this phase are executed. If one execution fails, all the build fails. If we need to extend maven build, we just have to declare a new plugin, and bind it to a phase.

3.1. Dependency management

We do not want to have code duplicated. We all have, in our projects, references to other chunks of code, in other projects. We all have a copy of `functx.xsl`[2], copied from project to project ! As there is no common mecanism to resolve those kinds of links in XML world, the usual way to solve this is to copy the code from source project, into other project where we need it. Others reference a GIT commit from another project, and check out this commit into project. Git as such a mecanism, but even if code is not duplicated, files are, and may be modified in host project. We want to rely on existing code, that has been build accordingly to our requirements, so we need to have:

- a way to store in a repository all release artifacts that have been build
- a way to reference an artifact we want to use (according to usual designation method)
- a way to access to a resource in an artifact.

Maven has a way to uniquely identify an artifact:

```
(groupId:artifactId:version)
```

groupId represents a unique key to project, and is based on Java package naming conventions ; *artifactId* represents something that is build by a maven module ; it must be unique per groupId ; *version* represents the artifact version ; a version

[2]Priscilla Walmsley `functx.xsl`, http://www.xsltfunctions.com/xsl/download.html

that ends with `-SNAPSHOT` is a snapshot, and is not strictly bound to a commit in SCM. All other strings represent a release, which is supposed to be uniquely bound to a sole commit in SCM.

In a Java Maven project, when using an external libray is required, it's enough to declare a *dependency* in project descriptor, *`pom.xml`*. If we want to use Saxon-HE 9.8.0-7 in our artifact, we just have to declare:

```
<dependencies>
    <dependency>
        <groupId>net.sf.saxon</groupId>
        <artifactId>Saxon-HE</artifactId>
        <version>9.8.0-7</version>
    </dependency>
</dependencies>
```

When Maven builds project, Maven downloads artifacts from central repository, copy them into a local repository, and constructs a classpath based on dependencies listed in pom. Included dependencies may declare other dependencies, and a full classpath is construct, based on the full dependency tree.

All resources in all jars declared as dependencies are accessible through standard Java resource loading mecanism:
`getClass().getResource("/upper-case.xsl")`.

So, during build, Maven knows the location of jars pointed by dependencies ; they all are in local repository.

We decided to reference resources in external projects via the standard Maven dependency mecanism, and by constructing URIs that can point a resource in a dependency. Each dependency is associated to a custom *URI protocol* which is its `artifactId:/`. Then, the usual way to construct a path in URIs is used to point a resource.

- if we want to reference the `net.sf.saxon.Transform` class in Saxon-HE 9.8.0-7 dependency, we'd construct
  `Saxon-HE:/net/sf/saxon/Transformer.class`
- if we want to reference the `file-utils.xsl` in `(eu.els.common:xslLibrary:3.1.7)`, we'd use `xslLibrary:/file-utils.xsl`

As it is common to change a dependency version, version is not included in URI ; hence, when changing a dependency version, code is not impacted.

If we declare xf-sas dependency:

```
<dependency>
  <groupId>eu.lefebvre-sarrut.sie.xmlFirst</groupId>
  <artifactId>xf-sas</artifactId>
  <version>3.1.7</version>
</dependency>
```

...we may have xsl with imports based on this URI syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="#all"
    version="3.0">
  <xsl:import
    href="xf-sas:/xf-sas/efl/mem2ee.alphamem.xsl"/>
```

At `initialize` Maven phase, so in very beginnning of build lifecycle, we do use a *catalogBuilder-maven-plugin*[3] that generates a catalog, based on dependencies declared in pom.xml. This catalog is generated at each build, so always denotes dependencies declaration available in project descriptor. It declares `rewriteURI` and `rewriteSystem` entries, that maps protocols to jar locations.

The catalog includes all dependencies that do contains XML resources, but also other dependencies, including the ones that do not concern XML processing ; this could be filtered in a future enhancement to make catalogs lighter.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog PUBLIC
  "-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN"
  "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
  <catalog
    xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <rewriteURI uriStartString="xsl-sas:/"
      rewritePrefix=
    "jar:file:/home/cm/.m2/repo/eu/els/xf-sas/1.2.1/xf-sas-1.2.1.jar!/"/>
    <rewriteSystem uriStartString="xsl-sas:/"
      rewritePrefix=
    "jar:file:/home/cm/.m2/repo/eu/els/xf-sas/1.2.1/xf-sas-1.2.1.jar!/"/>
</catalog>
```

This catalog is then used by all XML tools, including Maven plugins that do manipulate XML files. We have choosen to have, in all projects, such a catalog, named `catalog.xml`, at project's root. We can then define in oXygen, that we have to use a catalog named `${pdu}/catalog.xml` ; this allows oXygen to access all resources we reference in our code, including resources located into jar files, thanks to Java supporting URI like `jar:file:/ path/ to/ library.jar! / resource/to/file.xsl`

Developing in oXygen is the main reason of resolving resources to jar files, as oXygen is not maven aware, and do not loads dependency jars in project classpath. When running in Maven, all resources are in classpath, and could be directly access from their part-part of URI, in classpath.

[3]https://github.com/cmarchand/maven-catalogBuilder-plugin

This kind of resource references via URI mecanisms is quite common in XML world, and referencing resources from external dependencies can be - and is - generalized to all types :

- DOCTYPE definitions, references to DTD via SYSTEM declarations (we do not use PUBLIC definitions)
- imported or included XSL
- XML schema imports, includes, redefinitions
- RelaxNG, NVDL
- XQuery import module namespace

We have choosen to use a protocol based only on `artifactId:/`. But catalog-Builder-maven-plugin is able to use others patterns, based on groupId and artifactId, with a user-defined syntax.

3.2. Unit tests

Once dependency management solved, we tried to run unit tests each time a build is performed. We do use XSpec as a unit test framework. When we started this work, two XSpec Maven plugins were available: one written by Adam Retter, one by Daisy Consortium, maintained by Romain Deltour, both open source and available on Github.

Maven has a standard way to organize directories in a project:

```
.
└── src
    ├── main
    │   └── java
    └── test
        └── java
```

We decided that our XSL code will be put under `src/main/xsl`, and XSpec files under `src/test/xspec`. So, our project structure is always :

```
.
└── src
    ├── main
    │   └── xsl
    └── test
        └── xspec
```

XSpec Maven Plugin looks recursively in `src/test/xspec` for `*.xspec` files. Each file is executed, accordingly to XSpec implementation, and a report is generated. In Maven, all generated files are generated in `target/` directory. XSpec report file are generated in `target/xspec-report`. If one XSpec unit test fails, all the plugin execution fails, *test* phase fails, and build fails. Plugin is bound to test phase, but can also be bound to *integration-tests* if required.

Plugin has been largely enhanced to support catalogs, to allow to choose which Saxon product to use to perform tests, and reporting has been changed to be more useful. XSpec for XQuery support was not in Adam's Retter release, and has just been added ; but XSpec for Schematron support si still a work in progress.

As far as all unit tests do not succeed, we are not able to publish a release.

XSpec Maven Plugin[4] code has moved to XSpec organization, and is now maintained by the same team that maintains XSpec. XSpec implementation is now available as a Maven artifact, and this allows to deploy quickly XSpec corrections into XSpec Maven plugin. There is still some job to do: XSpec Maven Plugin is not able to run XSpec on Schematron, and JUnit reports are not generated.

3.3. Code generation

We have a grammar (RelaxNG) that has different distributions: one very strict, the other one 'lighter'. The lighter can easily be generated from the strict one, by applying a simple transformation. Instead of duplicating code, the strict grammar is released as an artifact, from its source code, but the lighter is generated from the strict one.

A new project declares a dependency on the strict grammar artifact, embeds XSL to transform strict grammar source files. At *generate-sources* phase, XSL are applied on strict source files, and this generates the 'lighter' grammar. Then generated sources are packaged, and deployed as a new artifact, with no source duplicated, or manually modified. We are able to distribute the ligther grammar as an artifact, without any source code, except the transformer.

We do use Maven XML Plugin[5] to apply XSL on RelaxNG source code, and to produce the 'lighter' grammar. It allows to embed Saxon as a XSLT 2.0 processor, and supports catalogs, so no enhancement were required to use this plugin.

At this time, we do not have a framework to perform unit tests on RelaxNG, but this could be done with other frameworks, like XMLUnit[6]. Job has not be done, due to lack of resources, but this is technically possible, and could easily be embeded in a maven plugin, bound to test phase.

Maven XML Plugin may be used to apply transformation on any XML source document, and is very suitable for generating sources.

[4]https://github.com/xspec/xspec-maven-plugin-1/
[5]http://www.mojohaus.org/xml-maven-plugin/
[6]http://www.xmlunit.org/

3.4. Source code documentation

Java has a standard way to produce source code documentation: javadoc. This system is very popular, and has been adapted to various programming languages. oXygen has defined a grammar to add documentation to XSL, xd-doc[7]

oXygen provides tools to generate a developer-oriented documentation, in an HTML format ; unfortunately, this tool is not open source, and could not be used directly. We have created a xslDoc Maven Plugin[8] that generates XSL documentation. This plugin is a report plugin, and can be added to Maven site reports. When you ask Maven to generate project's site, XSL documentation is added to project's site.

We have also created such a plugin for XQuery documentation[9], based on xquerydoc[10]. XQuery documentation is also generated as a site report when plugin is declared in pom.

```
<reporting>
  <plugins>
    <plugin>
      <groupId>top.marchand.xml</groupId>
      <artifactId>xslDoc-maven-plugin</artifactId>
      <version>0.6</version>
    </plugin>
    <plugin>
      <groupId>top.marchand.xml</groupId>
      <artifactId>xquerydoc-maven-plugin</artifactId>
      <version>0.1</version>
    </plugin>
  </plugins>
</reporting>
```

XQuery files are expected to be located in `src/main/xquery`.

3.5. Compiling

It's common that distributed code files differs from source code file. We could consider that transforming a source code file to a distributed code file is the compilation, whatever transforming is. Having distributed code transform, or "obfuscated", guarantees that it will never be modified in production environments. Mainly, we were interested in XSLT compilation. Thanks to XSL 3.0, we are now able to distribute compiled XSL files.

[7]https://www.oxygenxml.com/doc/versions/19.1/ug-editor/topics/XSLT-Stylesheet-documentation-support.html
[8]https://github.com/cmarchand/xslDoc-maven-plugin
[9]https://github.com/cmarchand/xqueryDoc-maven-plugin
[10]https://github.com/xquery/xquerydoc

When compiling an XSL file that declares imports and includes, only one file is generated, all imports and includes are merged into the one that will be run. You have less files to distribute, and you will not forget to distribute a dependency file.

Saxon-EE is able to compile XSL files. We've created a XslCompiler Maven plugin[11] that relies on Saxon-EE and that compiles XSL files from sources to `target/` directory. This plugin is still under evaluation, mainly for unit tests considerations.

Most of Xsl files are included or imported, but never directly run. Those files should not be compiled, i.e. should not have a distributed form ; this plugin requires developer to declare which Xsl source files must be compiled.

We have a problem in distributing compiled files : XSpec tests source files – uncompiled XSL files. XSpec implementation generates a XSL that includes the XSL we want to unit-test. Then this XSL is compiled (saved or not), and run. So, the compiled file that is tested is not the one that will be distributed.

Investigations on how to solve these problems have not been done ; but we could think to reference compiled XSL from XSpec file – if implementation allows it – or compiling generated XSpec XSL to a compiled form, and then running it ; all of this require talks with Saxonica, to know what can be done or not, what we can rely on, or not.

3.6. Packaging

Common packaging for a maven artifact is a jar file, that embeds only files from the project. The wide majority of our projects are published as jar files into repository manager. But these kind of jar files are not very suited to be deployed on a server. We have projects, which are only Java projects, that embeds a XSLT processor, a XProc processor, or gaulois-pipe, and all XML artifacts needed. Thoses projects delivers a jar file with all dependencies included, with a main-class entry in manifest, that allows to be run simply : `java -jar full-program-1.3.7.jar`. Such a jar is autonomous, and is very easy to deploy on a server. In such a jar, all resources from all dependencies are put in the same jar. Catalog needs to be recreated for such a jar, as all resources are directly in classpath, and not split into many jar files. We use rewriteURI and rewriteSystem to map all artifact based protocols to a *`cp:/`* protocol, cp for classpath. We have written a Java protocol handler for this protocol, which load resources directly from classpath. This can only be used in a JVM where the whole resources are in classpath ; when developing in oXygen, this is not true, that's why we map protocols to dependency jar files.

[11]https://github.com/cmarchand/xslcompiler-maven-plugin

4. Future

Dependency management now works perfectly, and do not need enhancements anymore. Maybe some more entries in generated catalog file, but this should be the limit.

Documentation needs to be beautified ; mainly XSL documentation. oXygen has really done a great job, and it's difficult to have something as beautifull as they produce, under a free and open source license. We still have some bugs, mainly on components identification, with `@use-when` attribute.

We are not able to generate documentation for grammars : mainly XML Schema and RelaxNG, and others like Schematron and NVDL ; we have some ideas, but not enough resources to put them in a useful plugin.

We hope other developers will publish their libraries as Maven artifacts ; hence, we could use them as dependencies ; today, we must put a lot of pom configuration to download source files (or zip files, or anything else), package them into an artifact, and deploy it into each local repository. It's not very easy, and requires Maven specialists. We would be very interested if skeleton organization, for example, were able to provide skeleton implementation as a maven artifact.

A similar approach could be used for a gradle build. Gradle also allows to declare a list of dependencies. It should be easy to write a plugin that generates a catalog. But it should probably require to write as many plugins as we had to for Maven.

Glossary

aertifact	An artifact is something that is either produced or used by a project. Examples of artifacts produced by Maven for a project include: JARs, source and binary distributions, WARs. Each artifact is uniquely identified by a group id and an artifact ID which is unique within a group.
groupId	A group ID is a universally unique identifier for a project. While this is often just the project name (eg. commons-collections), it is helpful to use a fully-qualified package name to distinguish it from other projects with a similar name (eg. org.apache.maven, net.sf.saxon, top.marchand.maven).
artifactId	An artifact ID is a unique identifer of an artifact within a group ID.

version
: The version of an artifact. Usually, there are snaphsot versions, which ends with `-SNAPSHOT`, and that denote a work in progress artifact ; all other version are release, and commit how produced this version is strictly known.

dependency
: A typical Java project relies on libraries to build and/or run. Those are called “dependencies” inside Maven. Those dependencies are usually other projects’ JAR artifacts, but are referenced by the POM that describes them.

pom.xml
: The file where project build is described. pom is for Project Object Model. It defines which dependency are required, which plugins are used to build project, and their configuration. See https://maven.apache.org/pom.html.

During XML Prague 2018 talk, Matthieu has made a demo based on this project : https://github.com/mricaud/xml-prague-2018-demo_myXMLproject

Varieties of XML Merge: Concurrent versus Sequential

Tejas Pradip Barhate
DeltaXML Ltd
<tejas.barhate@deltaxml.com>

Nigel Whitaker
DeltaXML Ltd
<nigel.whitaker@deltaxml.com>

Abstract

Merging the XML documents is a particularly tricky operation but is often required to consolidate or synchronize two or more independent edit paths or versions. As XML tools become more powerful and able to handle many of the peculiarities of real data, so the possibility of achieving a genuine, intelligent merge of XML data sets becomes a reality. The complexity of XML places demands on tools to work intelligently in order to preserve the essential structure of the original document and also represent the changes.

This paper discusses the different varieties of merge for XML. Merging multiple derivatives of a single ancestor (concurrent merge) may be the most obvious application, but there is also a need for a sequential merge when a document has been passed around between two or more authors in a sequential manner. Another important, and perhaps less well understood, application is 'graft', where the changes between two documents or data sets are applied to a third, different (though similar) document or data set.

There are of course similarities between these applications, but gaining an understanding of how they differ and where each is appropriate is necessary to make best use of automated processing of XML.

1. Introduction

Over the years, the XML is widely used in different fields and different merge solutions were proposed. But as the size and complexity of the XML files increases, it becomes difficult to manage the XML documents or data sets.

The need for XML Merge arises from scenarios like document reviews, concurrent changes, document revision history, or even, combining XML data sets. It is evident that different problems exist in different usage scenarios, and there is a need for a flexible solution, which can be adapted to different needs.

There are number of solutions available to merge, synchronise or combine number of XML documents or data file into one. But this paper introduces the merges which not only combine number of XML documents together by preserving the structure of original document but also aims to show the changes which can be processed further.

This paper also talks about various types of XML merge use cases and discusses how these different scenarios can be represented into single intermediate XML file which can be used to process the changes.

1.1. Background

A n-way merge algorithm has been developed to address the issues and varieties of merge processes and use cases discussed in this paper. It is the third algorithm developed over several years starting with [1]. As a new 'clean sheet' implementation we decided to implement n-way merge as it provides a general basis for a number of related use-cases which we aim to describe in this paper.

2. Various type of merges

This section presents the different varieties of merge along with their similarities and differences. They differ in terms of frame of reference, alignment and change representation. But any change can be represented using the same format.

2.1. Concurrent Merge

Concurrent merge recombines multiple XML files with their common ancestor, analysing their structure and running custom rules to either merge or explicitly mark-up the differences. Its algorithms work through each of the files in turn, examining their structure to match-up all the corresponding elements with the original.

Once all the differences have been identified, changes are represented in a structured intermediate delta XML file where the structure of the delta file is similar to the ancestor. The changes can be described by three operations: modify, add and delete. Because we can consider the ancestor version to be older than the other edits in the merge, the concepts of add and delete are defined relative to the ancestor version. This choice then leads to the following definitions for change categorisation.

- **add**: Something that does not exist in the ancestor version. The item may be added by one or more of the other versions.
- **delete**: Something that exists in the ancestor version, but is missing in one or more of the other versions.

2.2. Sequential Merge

The sequential merge merges one or more sequentially edited XML documents. One of the important characteristics of the sequential merge is that there is a clearly-defined order of editing. The order of editing provides the temporal frame of reference and so the concepts of add and delete are defined relative to the order of editing. This choice then leads to the following definitions for change categorisation.

- **add**: Something that does not exist in the previous version. When something is added it has never been seen before.
- **delete**: Something that exists in the previous version, but is missing in this version. As soon as something is deleted, then it cannot be added back again, rather if the same item appears again then a new version is created, with no relationship with the deleted item.

3. Concurrent Vs Sequential

3.1. Attribute: deltaxml:deltaV2

The format used for merge representation has a deltaxml:deltaV2 attribute which contains contain a sequence of one or more 'version identifiers' joined by the '=' character or '!=' character-pair. The document versions with different content are separated by the '!=' whereas The document versions with same content are separated by the '=' character-pair. The versions at the certain level with same content can be considered as equality groups. The versions within equality groups and between equality groups (i.e. groups of versions separated with '!=') are ordered according to an ordering sequence (order in which they were added to merge).

The one major difference between the value of deltaxml:deltaV2 attribute in sequential and concurrent merge is the order of the versions. In sequential merge, the versions in this attribute always occur in the order whereas in concurrent merge they can appear in any order provided the first version in the attribute is ancestor (if the element exists in ancestor). This conforms with the behaviour of sequential and concurrent merge i.e. the sequential merge takes documents which are derivatives of its previous version whereas concurrent merge takes documents which are derived from an ancestor version.

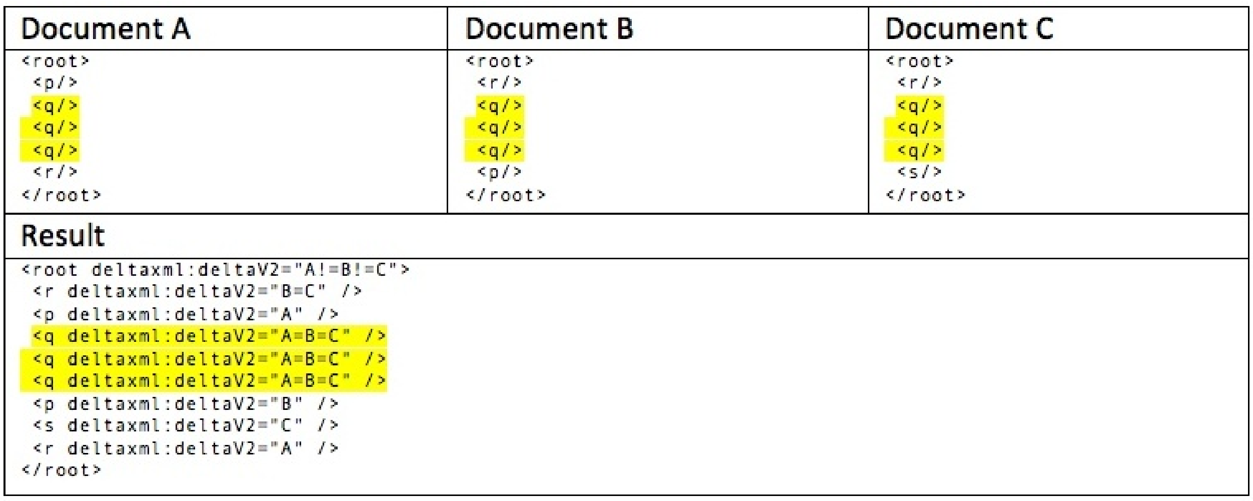
3.2. Structure and Alignment

As mentioned in previous sections, the merge process merges the XML files, taking account of the tree structure of the files and identifying corresponding elements in the files. These corresponding elements must have the same element local name and namespace and should also have corresponding parent elements. The root elements of the files must have the same local name and namespace.

We maintain the alignment at each level in the tree structure between the files by determining the longest common subsequence of corresponding elements. The alignment algorithm gives precedence to elements that are exactly equal over those that have just the same element name and namespace. While merging, the algorithm also takes into account the similarity of the text content and aligns elements based on this similarity.

The merge process accepts the inputs in some order and this order gets recorded into an attribute on the root element of the merge result. For concurrent merge, the version is first aligned with the common ancestor and this alignment will take precedence over alignment between this version and other versions previously loaded into the merge. Whereas in sequential merge, as each successive file is loaded into the merge, the version is aligned with the previous version.

In the following example, you will see that the three <q> elements have been matched although the elements before and after differ.



Document A

```
<root>
 <p/>
 <q/>
 <q/>
 <q/>
 <r/>
</root>
```

Document B

```
<root>
 <r/>
 <q/>
 <q/>
 <q/>
 <p/>
</root>
```

Document C

```
<root>
 <r/>
 <q/>
 <q/>
 <q/>
 <s/>
</root>
```

Result

```
<root deltaxml:deltaV2="A!=B!=C">
 <r deltaxml:deltaV2="B=C" />
 <p deltaxml:deltaV2="A" />
 <q deltaxml:deltaV2="A=B=C" />
 <q deltaxml:deltaV2="A=B=C" />
 <q deltaxml:deltaV2="A=B=C" />
 <p deltaxml:deltaV2="B" />
 <s deltaxml:deltaV2="C" />
 <r deltaxml:deltaV2="A" />
</root>
```

Figure 1. Alignment

3.3. Alignment using keys

The merge can use key values to identify the corresponding elements in the inputs. Alignment of elements with the same namespace, local name and key will take precedence in the alignment process over other alignment criteria. Elements with different keys will not be considered to correspond, and therefore keys can be used to prevent elements being aligned as corresponding.

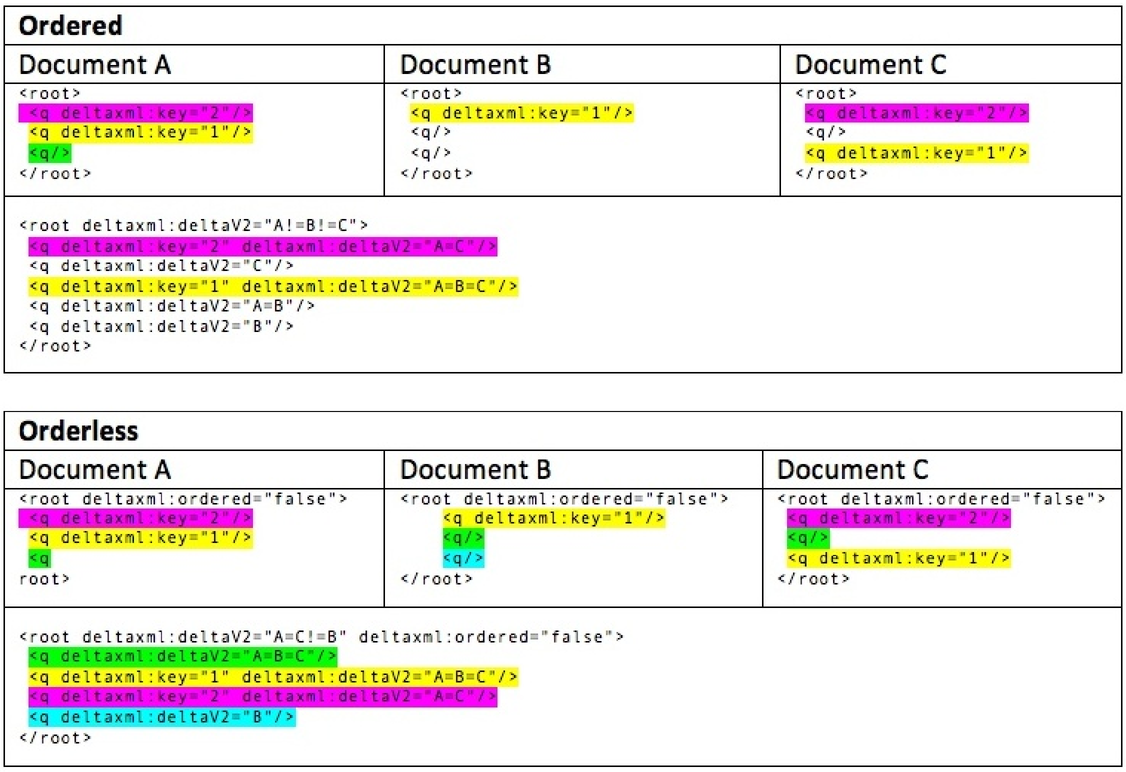
3.3.1. Ordered and Orderless

While applying keys it is significant to know if the order of element within a container/parent element is significant or not. The document with set of instructions can be considered as ordered while the document having a list of information about persons might be orderless. In the ordered comparison, the relative position of a keyed element is important, whereas the orderless elements are members of a set. Thus, any extracted version from the ordered merge will preserve the order.

The following examples demonstrate the difference between ordered and orderless keying and how they differ in concurrent and sequential merge scenario. The merge algorithm internally uses 'deltaxml:key' attribute as a key as shown in the examples below.

3.3.2. Concurrent Keying Vs Sequential Keying

Due to difference of how each version is modified in concurrent and sequential merge, the keyed result also differs. For example, in the following example ‘q’ element with key ‘2’ exists in version A and C. These elements align properly in concurrent merge; however they must not align in sequential merge. This results into key duplication in the sequential result if the element with same key is deleted and a new element with same key is added later. Its is very difficult to avoid this situation in real life as the editor does not know what was deleted in previous versions. However, these elements with same key can be differentiated using their deltaxml:deltaV2 attribute.



Ordered

Document A
```
<root>
 <q deltaxml:key="2"/>
 <q deltaxml:key="1"/>
 <q/>
</root>
```

Document B
```
<root>
 <q deltaxml:key="1"/>
 <q/>
 <q/>
</root>
```

Document C
```
<root>
 <q deltaxml:key="2"/>
 <q/>
 <q deltaxml:key="1"/>
</root>
```

```
<root deltaxml:deltaV2="A!=B!=C">
 <q deltaxml:key="2" deltaxml:deltaV2="A=C"/>
 <q deltaxml:deltaV2="C"/>
 <q deltaxml:key="1" deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:deltaV2="A=B"/>
 <q deltaxml:deltaV2="B"/>
</root>
```

Orderless

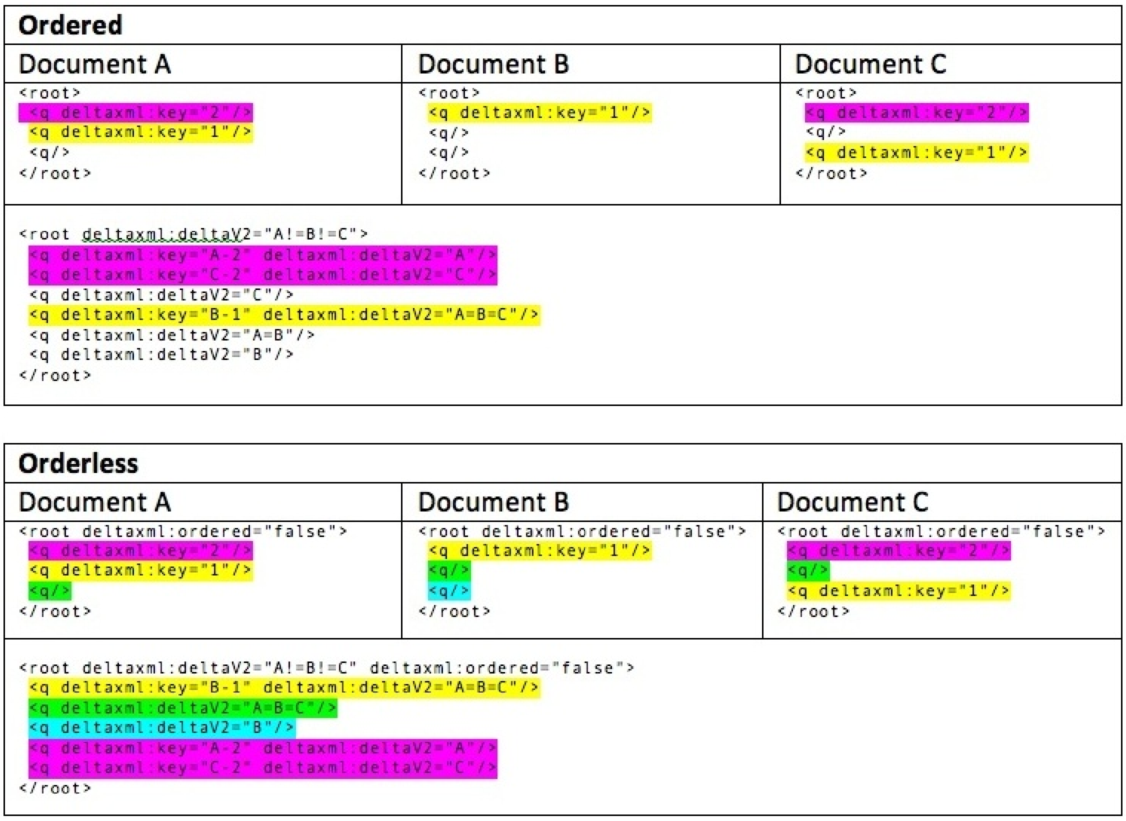
Document A
```
<root deltaxml:ordered="false">
 <q deltaxml:key="2"/>
 <q deltaxml:key="1"/>
 <q
root>
```

Document B
```
<root deltaxml:ordered="false">
     <q deltaxml:key="1"/>
     <q/>
     <q/>
</root>
```

Document C
```
<root deltaxml:ordered="false">
 <q deltaxml:key="2"/>
 <q/>
 <q deltaxml:key="1"/>
</root>
```

```
<root deltaxml:deltaV2="A=C!=B" deltaxml:ordered="false">
 <q deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:key="1" deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:key="2" deltaxml:deltaV2="A=C"/>
 <q deltaxml:deltaV2="B"/>
</root>
```

Figure 2. Concurrent Merge Keying:



Ordered

Document A

```
<root>
 <q deltaxml:key="2"/>
 <q deltaxml:key="1"/>
 <q/>
</root>
```

Document B

```
<root>
 <q deltaxml:key="1"/>
 <q/>
 <q/>
</root>
```

Document C

```
<root>
 <q deltaxml:key="2"/>
 <q/>
 <q deltaxml:key="1"/>
</root>
```

```
<root deltaxml:deltaV2="A!=B!=C">
 <q deltaxml:key="A-2" deltaxml:deltaV2="A"/>
 <q deltaxml:key="C-2" deltaxml:deltaV2="C"/>
 <q deltaxml:deltaV2="C"/>
 <q deltaxml:key="B-1" deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:deltaV2="A=B"/>
 <q deltaxml:deltaV2="B"/>
</root>
```

Orderless

Document A

```
<root deltaxml:ordered="false">
 <q deltaxml:key="2"/>
 <q deltaxml:key="1"/>
 <q/>
</root>
```

Document B

```
<root deltaxml:ordered="false">
 <q deltaxml:key="1"/>
 <q/>
 <q/>
</root>
```

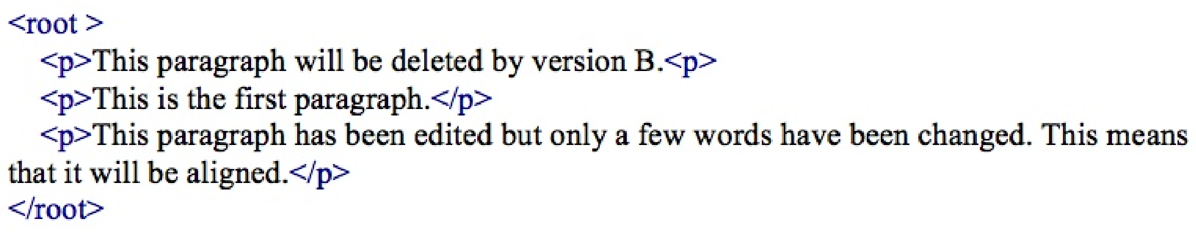
Document C

```
<root deltaxml:ordered="false">
 <q deltaxml:key="2"/>
 <q/>
 <q deltaxml:key="1"/>
</root>
```

```
<root deltaxml:deltaV2="A!=B!=C" deltaxml:ordered="false">
 <q deltaxml:key="B-1" deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:deltaV2="A=B=C"/>
 <q deltaxml:deltaV2="B"/>
 <q deltaxml:key="A-2" deltaxml:deltaV2="A"/>
 <q deltaxml:key="C-2" deltaxml:deltaV2="C"/>
</root>
```

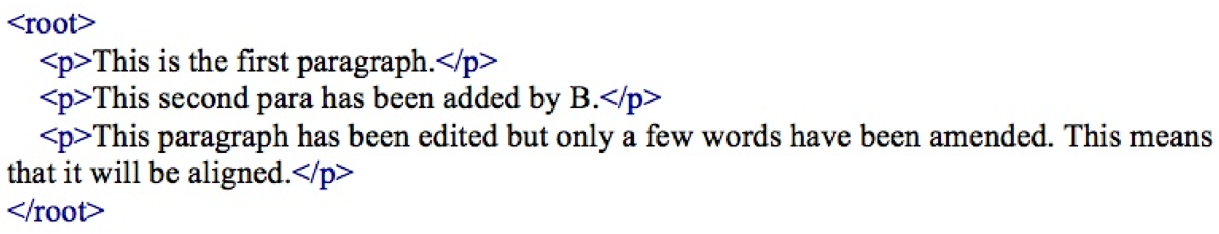
Figure 3. Sequential Merge keying:

3.4. Example



```
<root >
   <p>This paragraph will be deleted by version B.<p>
   <p>This is the first paragraph.</p>
   <p>This paragraph has been edited but only a few words have been changed. This means
that it will be aligned.</p>
</root>
```

Figure 4. Document A



```
<root>
   <p>This is the first paragraph.</p>
   <p>This second para has been added by B.</p>
   <p>This paragraph has been edited but only a few words have been amended. This means
that it will be aligned.</p>
</root>
```

Figure 5. Document B

```
<root>
   <p>This paragraph will be deleted by version B.<p>
   <p>In the C document, this initial introduction has been added.</p>
   <p>This was the first paragraph.</p>
   <p>This paragraph has been edited but only a few words have been changed. This means
that it will be aligned.</p>
</root>
```

Figure 6. Document C (Concurrent Editing)

Version A and B will be same as above during sequential edits, however C will differ as it will not have a deleted para by version B.

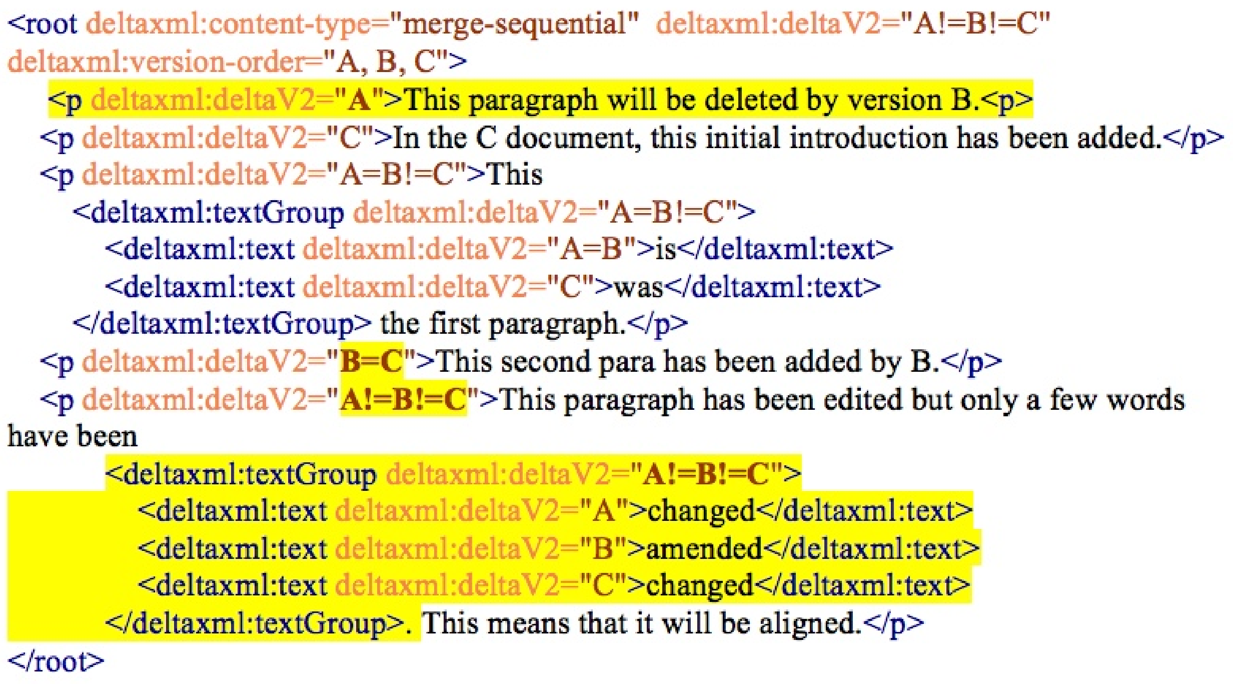
```
<root>
   <p>In the C document, this initial introduction has been added.</p>
   <p>This was the first paragraph.</p>
   <p>This paragraph has been edited but only a few words have been changed. This means
that it will be aligned.</p>
</root>
```

Figure 7. Document C (Sequential Editing)

The following concurrent and sequential merge results are highlighted to show the difference between two.

```
<root deltaxml:content-type="merge-concurrent" deltaxml:deltaV2="A!=B!=C"
deltaxml:version-order="A, B, C">
   <p deltaxml:deltaV2="A=C">This paragraph will be deleted by version B.<p>
   <p deltaxml:deltaV2="C">In the C document, this initial introduction has been added.</p>
   <p deltaxml:deltaV2="A=B!=C">This
      <deltaxml:textGroup deltaxml:deltaV2="A=B!=C">
         <deltaxml:text deltaxml:deltaV2="A=B">is</deltaxml:text>
         <deltaxml:text deltaxml:deltaV2="C">was</deltaxml:text>
      </deltaxml:textGroup> the first paragraph.</p>
   <p deltaxml:deltaV2="B">This second para has been added by B.</p>
   <p deltaxml:deltaV2="A=C!=B">This paragraph has been edited but only a few words
have been
         <deltaxml:textGroup deltaxml:deltaV2="A=C!=B">
            <deltaxml:text deltaxml:deltaV2="A=C">changed</deltaxml:text>
            <deltaxml:text deltaxml:deltaV2="B">amended</deltaxml:text>
         </deltaxml:textGroup>. This means that it will be aligned.</p>
</root>
```

Figure 8. Concurrent Merge Result



```
<root deltaxml:content-type="merge-sequential"  deltaxml:deltaV2="A!=B!=C"
deltaxml:version-order="A, B, C">
   <p deltaxml:deltaV2="A">This paragraph will be deleted by version B.<p>
   <p deltaxml:deltaV2="C">In the C document, this initial introduction has been added.</p>
   <p deltaxml:deltaV2="A=B!=C">This
      <deltaxml:textGroup deltaxml:deltaV2="A=B!=C">
         <deltaxml:text deltaxml:deltaV2="A=B">is</deltaxml:text>
         <deltaxml:text deltaxml:deltaV2="C">was</deltaxml:text>
      </deltaxml:textGroup> the first paragraph.</p>
   <p deltaxml:deltaV2="B=C">This second para has been added by B.</p>
   <p deltaxml:deltaV2="A!=B!=C">This paragraph has been edited but only a few words
have been
         <deltaxml:textGroup deltaxml:deltaV2="A!=B!=C">
            <deltaxml:text deltaxml:deltaV2="A">changed</deltaxml:text>
            <deltaxml:text deltaxml:deltaV2="B">amended</deltaxml:text>
            <deltaxml:text deltaxml:deltaV2="C">changed</deltaxml:text>
         </deltaxml:textGroup>. This means that it will be aligned.</p>
</root>
```

Figure 9. Sequential Merge Result

The highlighted part in above examples shows the difference between the two results. The first paragraph is deleted by 'B', so it never appears again in the sequential merge, however it can exist in 'C' in the concurrent merge. Similarly, in the last paragraph, there is text change. For the sequential merge, even though the 'A' and the 'C' texts are same, there is a deletion in between and so it is shown as three changes rather than a two-way change. The main purpose of the above examples is to understand what kind of inputs we have and how we want to see the result. It is possible to generate both concurrent and sequential merge results from any inputs, but one of these is more likely to be the correct choice for a particular set of data. The usefulness of the result will depend on how the XML inputs have been modified and what type of merge is applied.

3.5. Merge Analysis

The merge result accurately describes the contributions of all of the input files. The deltaxml:deltaV2 attribute describes the which inputs contribute to the result and whether they are identical or different subtrees at that point in the tree. If we analyse the sequential merge, and compare it with an ordered list of the inputs to the merge process, then the first version in deltaxml:deltaV2 adds the content and first missing version, if any, deletes the content. So, analysing the sequential result is simple and straight forward. However, the analysis of concurrent merge can be harder, especially when there are more than three inputs. This is mainly because to determine whether something has been added or deleted we need to consider the number of versions at the point of interest, the number of versions in the parent and where the ancestor, or first version in the delta is present.

However, human understanding and subsequent processing is simplified if these deltas are classified to describe the types of change which they represent. This is achieved by adding annotations such as add, delete or modify for each change. The following table describes the different types of changes using deltaxml:deltaV2 where order of version is: Ancestor, A, B, C, D.

Table 1. Change Annotations

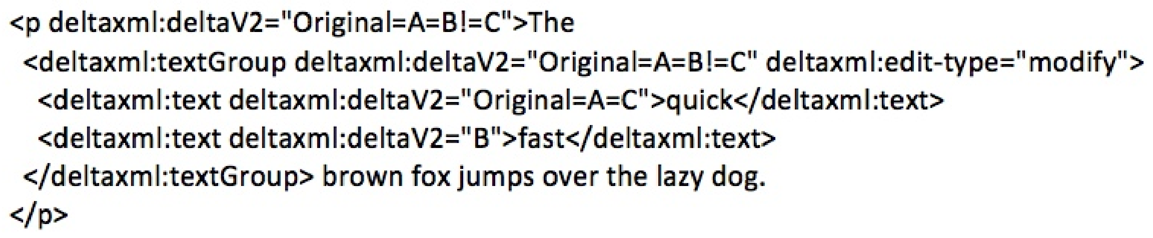
Change Type	Change Representation	Analysis
Modification	Ancestor=A=B!=C=D	All versions present, with at least one change (represented by !=)
Addition Or Addition and modification	B or B=C!=D	Ancestor not present in element being considered
Deletion Or Deletion and modification	Ancestor=A=C=D or Ancestor=A!=C=D	Ancestor is present in element being considered and fewer version than parent

3.6. Concurrent Merge Rule's Based Processing

The result of concurrent merge contains complete representations of all of the contents in the merge inputs. However, the automatic acceptance of the changes such as additions, deletions or modifications into a result is a common use-case for line-based merge algorithms and similar processing is useful with the XML based algorithms.

The motivation for developing a rule-based processing system follows from the design of line-based merge or 'diff3' algorithms used in software version control systems. These systems typically accept non-conflicting changes, so that lines that are changed but not in conflict are merged into the result.

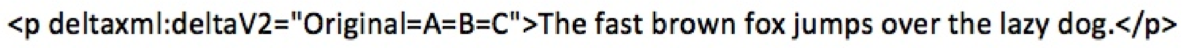
The rule-based processing provides a set of rules to determine which types of change are automatically applied. Without any user-specified processing rules the processing engine will, by default follow the example of diff3 and process simple add changes so that the content is added. Similarly, simple delete changes are removed from the result and simple (leaf) modifications are applied. The processing rules allow control over which changes are displayed to the user (for example, for subsequent interactive checking or resolution). Another way of thinking about the display rules is that they control which changes are not automatically applied or converted.



```
<p deltaxml:deltaV2="Original=A=B!=C">The
  <deltaxml:textGroup deltaxml:deltaV2="Original=A=B!=C" deltaxml:edit-type="modify">
    <deltaxml:text deltaxml:deltaV2="Original=A=C">quick</deltaxml:text>
    <deltaxml:text deltaxml:deltaV2="B">fast</deltaxml:text>
  </deltaxml:textGroup> brown fox jumps over the lazy dog.
</p>
```

Figure 10. Result

The above is an example of a modification which can be rule processed. The default action of the rule processing system would be to accept the simple change. The corresponding output would then be:



```
<p deltaxml:deltaV2="Original=A=B=C">The fast brown fox jumps over the lazy dog.</p>
```

Figure 11. Rule Processed Result

The rule based processing plays a vital role in the concurrent merge applications described in later sections.

4. Applications

4.1. Sequential Merge Applications

4.1.1. Travelling Draft

Negotiating a written contract where each party is making changes to consecutive versions can be a very complicated process. The usual way to do this is to track changes but these soon become difficult to understand and there is no guarantee that all the changes have been tracked. Therefore it is advantageous to use sequential merge to generate a 'true' travelling draft, i.e. one that does contain all the sequential changes. Each successive version can be added to this travelling draft to create a record of the changes.

4.1.2. Revision History

Content Management Systems (CMS) maintain a record of each successive version of a document. Any two versions can be compared to see what has been changed, but it may be more useful to gain an oversight of all the changes by using sequential merge to merge multiple versions together to see the history of revisions. Another advantage of constructing a sequential merge is that minor versions can be left out to reduce the complexity.

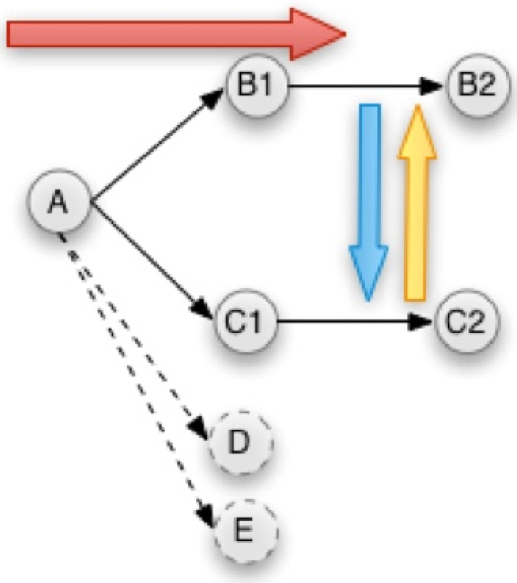
4.2. Three Way Concurrent Merge Applications

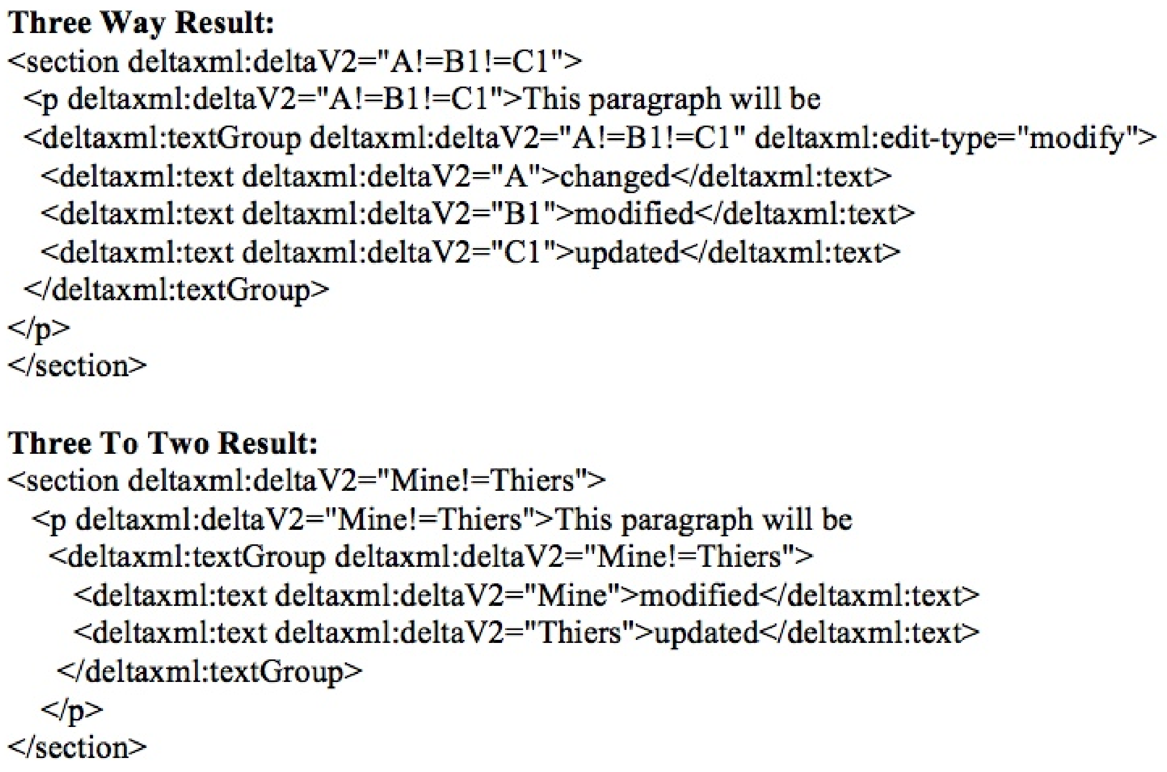
4.2.1. Three To Two Merge

Three To Two merge is an extension of a three-way concurrent merge. The representation of three-way conflicts is not supported by many XML editors. On the other hand, XML editors do support two-way change tracking and this is well understood by users. Therefore, if we could represent three-way merge conflicts in two-way change tracking this would provide a significant and useful simplification for users. This is possible without losing important information although some fine detail is lost, but it is much easier for a user to understand the result.

For this operation we need assume that all non-conflicting or 'simple' changes are accepted and changes between the ancestor (A in the diagram below) and the two inputs being merged (B1 and C1) are not as important as the differences between the two inputs themselves (B1 and C1).

Therefore, we can represent a conflicting change as an addition-deletion pair. For example, if we are merging C1 into the B branch (merge of A, B1, C1 and indicated with the yellow arrow, in order to generate B2) then a conflict is represented as a deletion of the conflicting content in B1 and addition of the conflicting content in C1. This has the characteristic that if all (conflicting) changes are rejected then we get the result of B1 unchanged where there is conflict, and if all (conflicting) changes are accepted then we get the result from C1 where there is conflict. This is simpler to explain to users, and we only need two-way change tracking to represent this.





Three Way Result:

```
<section deltaxml:deltaV2="A!=B1!=C1">
  <p deltaxml:deltaV2="A!=B1!=C1">This paragraph will be
  <deltaxml:textGroup deltaxml:deltaV2="A!=B1!=C1" deltaxml:edit-type="modify">
    <deltaxml:text deltaxml:deltaV2="A">changed</deltaxml:text>
    <deltaxml:text deltaxml:deltaV2="B1">modified</deltaxml:text>
    <deltaxml:text deltaxml:deltaV2="C1">updated</deltaxml:text>
  </deltaxml:textGroup>
</p>
</section>
```

Three To Two Result:

```
<section deltaxml:deltaV2="Mine!=Thiers">
  <p deltaxml:deltaV2="Mine!=Thiers">This paragraph will be
   <deltaxml:textGroup deltaxml:deltaV2="Mine!=Thiers">
     <deltaxml:text deltaxml:deltaV2="Mine">modified</deltaxml:text>
     <deltaxml:text deltaxml:deltaV2="Thiers">updated</deltaxml:text>
    </deltaxml:textGroup>
   </p>
</section>
```

Figure 12. Example

The example below illustrates this. The example shows that when representing the Three To Two result we need to consider what information is lost (we can only represent two of the three values) and how the remaining values are used. We lose the ancestor information in the process of conversion from a full three-way to a two-way representation. In the example, the C1 can be considered as 'their branch' (or 'Theirs') and B1 can be considered as 'my branch' (or 'Mine').

As well as the three way case illustrated above, another limitation of track-change systems based on XML Processing Instructions is that they typically cannot support nested change (for example an addition with contains modification or perhaps deletion further down the tree). Nested change is found in merge systems that are tree based and have three or more inputs. The diff3 algorithm used in software version control systems such as a git or mercurial are based on sequence of lines and therefore do not have to deal with nested change. Nested change introduces interesting possibilities of change representation and also user interfaces associated with change display and management [2].

4.2.2. Graft

Another variant for three-way concurrent merge is 'graft'. This is a way to propagate changes across similar files.

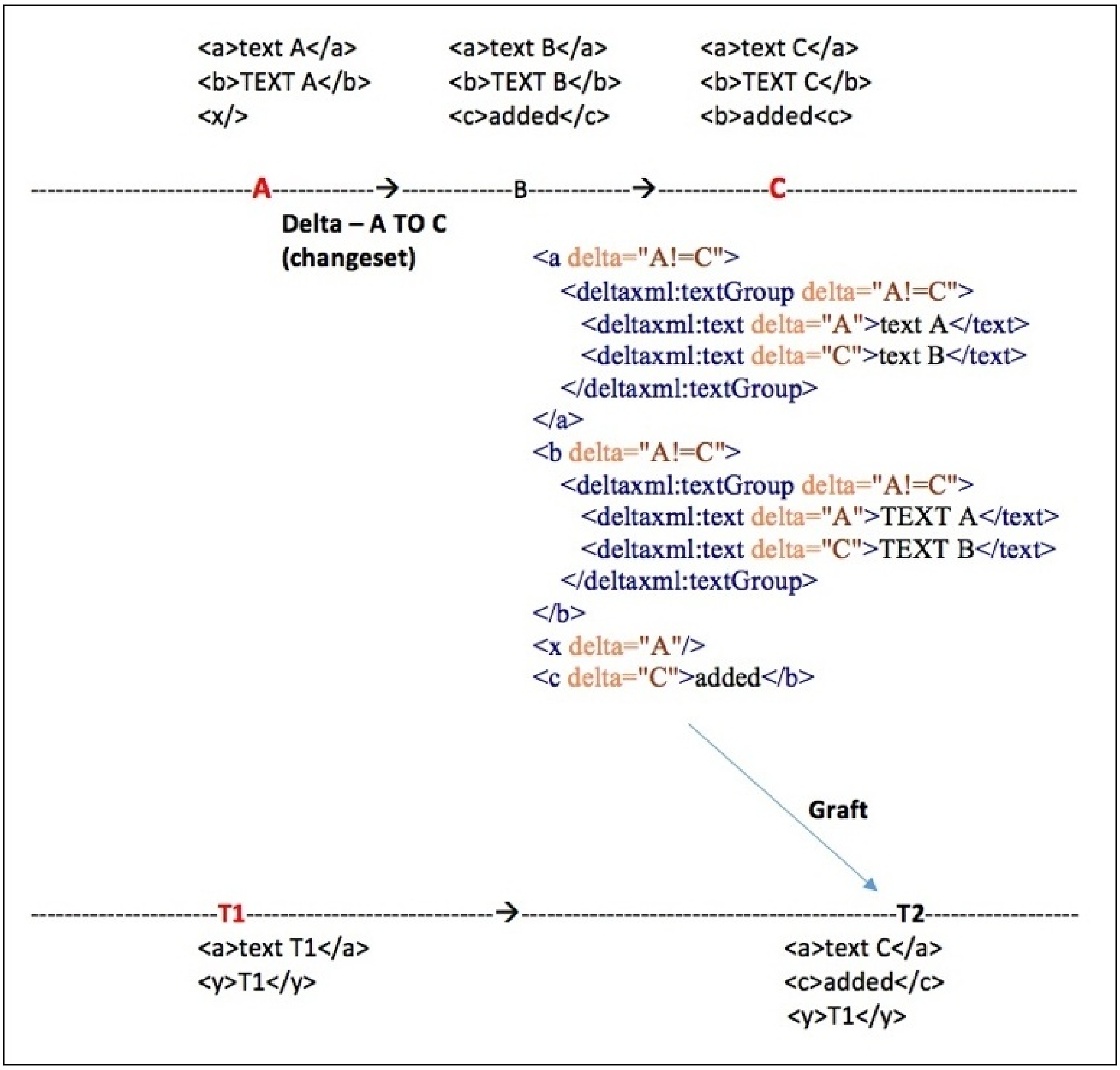
Another term used for graft is 'cherry pick'. This comes from version control systems where there are related branches and there is a need to cherry pick changes made between two versions in one branch and apply these to the data in another branch. This is not quite the same as a full three-way merge (merging two descendants of a common ancestor), but it is similar.

An XML delta file (produced by comparing two versions) represents a set of changes or changeset - so applying those changes to a target file is the way to perform a graft. So a graft executes all the relevant changes to the target file. By default, all the 'relevant' changes are applied. 'Relevant' here means that the data that is being changed is in the target file - if a change is made to data that does not appear in the target file, it is ignored.

Graft is probably more applicable to XML data files than XML documents. For example, if we have a master file with 500 names and addresses, and a subset with just 50 of them, we can apply changes made to the master file to the subset. In fact graft can also apply changes made in the subset to the master file. Similarly, we can take a list of changes to names and phone numbers and apply that to a related list of names, phone numbers and addresses. Or, again, the other way round (and then of course any changes to addresses will be ignored because there are no addresses in the target file).

It is a useful characteristic of the graft process if it can be made 'idempotent'. This means that a second application of the graft will leave the file as it was. This is advantageous not just because a second application may be made in error, but it handles the situation where some of the changes defined in the changeset have already been made in the target file.

Graft rules are subtly different from those for three-way merge. With three-way merge there is a common ancestor, i.e. a file from which the other two files are derived. This means we can detect the changes made in both branches and then merge these according to some set of rules. The rules can get complicated and there can be conflicts between the changes that have been made in the two derived files. Graft is a bit different, because there is no common ancestor. Therefore there is only the concept of changes made to one branch, which we want to apply to a target. There is no concept of changes that may have been made to the target (before we apply the graft), it is just there as a target data file. This is important because it means that if the target contains a subset or a superset of the data in the graft (or delta) file, this is acceptable because irrelevant changes are simply ignored. So we can have a whole set of related data files and apply changes made to any one to any of the others.



```
<a>text A</a>
<b>TEXT A</b>
<x/>
```

```
<a>text B</a>
<b>TEXT B</b>
<c>added</c>
```

```
<a>text C</a>
<b>TEXT C</b>
<b>added<c>
```

```
<a delta="A!=C">
  <deltaxml:textGroup delta="A!=C">
    <deltaxml:text delta="A">text A</text>
    <deltaxml:text delta="C">text B</text>
  </deltaxml:textGroup>
</a>
<b delta="A!=C">
  <deltaxml:textGroup delta="A!=C">
    <deltaxml:text delta="A">TEXT A</text>
    <deltaxml:text delta="C">TEXT B</text>
  </deltaxml:textGroup>
</b>
<x delta="A"/>
<c delta="C">added</b>
```

```
<a>text T1</a>
<y>T1</y>
```

```
<a>text C</a>
<c>added</c>
<y>T1</y>
```

Figure 13. XML Graft Example

The principle in above example is that delta- A to C contains all the changes, and each one is applied to T1 if it makes sense. So if a change is made to an object in A and there is a corresponding object in T1, then the change is applied. If not, the change is ignored.

5. Conclusion

In this paper we have discussed the different types of merge along with similarities and differences between them. They differ in terms of how the inputs are revised or modified and how they are aligned. With the help of a simple example it was demonstrated that even if the concurrent and sequential merge use the same format to represent the merge, the results produced are different. In addition to this, we also discussed the alignment using unique keys either to ensure the two elements do align with each other, or to prevent two elements from aligning with each other.

The examples and applications of concurrent and sequential merge should provide the useful information to make best use of automated processing of XML by analysing and resolving the merge result. XML provides a useful representation for defining and discussing these complex merge operations.

Bibliography

[1] Robin La Fontaine *Merging XML files: a new approach providing intelligent merge of XML data sets* Presented at XML Europe 2002.

[2] Nigel Whitaker *Understanding Changes in n-way Merge: Use-cases and User Interface Demonstrations* DChanges '14 - 2nd International Workshop on (Document) Changes: modeling, detection, storage and visualization. Proceedings of the 2014 ACM Symposium on Document Engineering.

[3] Tancred Lindholm *A Three-way Merge for XML Documents* Proceedings of 2004 ACM symposium on Document engineering.

Including XML Markup in the Automated Collation of Literary Text

Elli Bleeker
Huygens ING
<elli.bleeker@huygens.knaw.nl>

Bram Buitendijk
Huygens ING
<bram.buitendijk@huygens.knaw.nl>

Ronald Haentjens Dekker
Huygens ING
<ronald.dekker@huygens.knaw.nl>

Astrid Kulsdom
Huygens ING
<astrid.kulsdom@huygens.knaw.nl>

1. Introduction

XML plays a key role in humanities research. Scholars use it to express their understanding and interpretation of a text and to create textual models for further analysis. The affordances of the XML data model allow them to structure literary texts and to capture a wide range of textual phenomena, from physical characteristics of a historical document to the composition of a theatre play. By adding markup to the transcription, in other words by encoding it, scholars can make explicit their interpretation and their knowledge of a text. The encoding of literary texts generally results in text-centric XML files that consist of text data and XML markup elements.

The main challenge is that these encoded literary texts are neither fully ordered nor unordered data. They can be classified as "partially ordered XML" and require an entirely different way of parsing and processing than data-centric XML or text-centric XML. Traversing data-centric XML is relatively easy because it is unordered. Its structure is expressed by the schema, while the order of the elements is irrelevant. Data-centric XML contains properties and values, for example: when describing a person, we record their surname, first name, address and postal code. The order in which we list these properties does not contribute to the information: each individual statement stays true regardless of the order in which they are listed. The structure informs the queries. In data-centric XML queries can be exact, as in: "Give me the last name of the person living at Down-ing Street number 10." If multiple persons fit the description, the order in which they are listed in the result does not matter.

Standard text-centric XML, then, is in principle fully ordered. The value of the data is in the content (PCDATA), not in the structure. This content is flat (or "free") data and the order of the XML elements is crucial. Because of their fully ordered nature, they are parsed and processed differently than data-centric XML. Traversing text-centric XML means traversing the text from top to bottom, left to right, in order to display it or to transform it. In contrast to data-centric XML, queries on text-centric XML are rather inexact, because you are looking for a pattern of words. When querying text, the order in which the results are returned is relevant. This regards not only the order of the words within a result; the order of the individual search results is crucial as well. The text fragments that match the given patterns are listed in order of "best match".

Texts in the humanities, be they literary or historical or scholarly, are neither fully ordered nor unordered. For instance, a written text may contain internal variation in the form of revisions or additions in the margin. These variants are on the same level in the XML tree: they have the same rank and between them order is irrelevant. So while the text data is fully ordered, at the points in the text where variation occurs the file is unordered. Within the markup elements, however, the text data is again fully ordered. This combination of fully ordered and unordered makes text-centric XML of literary texts extremely challenging to parse and process in a satisfactory manner. At the same time it is paramount we confront these challenges, because it will provide us with a more realistic model of text, which in turn promotes better and original forms of research.

In the practice of literary text research XML files are often processed as plain text, which conveniently removes the need to tackle issues like overlap on a programmatic level. Since the structure of a tree can hold more information than plain text, the loss of information is inevitable. This concession is generally accepted, because the focus of literary text research is primarily on *text* and less on its structural features.

However, we propose a hypergraph data structure to process text-centric XML files. This approach removes the need to process text-centric XML files as plain text [and it discards the distinction between first class and second class objects]. At the same time it allows us to actually make use of the structural information represented by the XML markup. The scholarly knowledge contained by the XML markup, then, can be employed to improve (the outcome of) the processing.

The hypergraph data structure supports different forms of document modelling and processing.The present paper takes as point of departure an especially complex phenomenon, i.e. textual variation, and demonstrates in a step-by-step manner how a tool based on a hypergraph data model can address several challenges related to capturing textual variation in an XML environment.

The paper is organised as follows. First, we give a brief account of what it means to study literary text and text variation in a computational environment, and we outline some familiar and less-familiar issues that arise during text modelling. We expand upon the idea that text isn't linear and on the concept of collation, i.e. the comparison of text versions, which constitutes an important part of the study of textual variation. While it occurs in many different forms, textual variation can be divided into two main categories: multiple paths through a text, and variation in the textual structure. A tool that examines textual variation in an inclusive way therefore needs to be able to deal with variation in both categories. Section 3 begins with a detailed description of the hypergraph model for textual variation followed by a presentation of the tool HyperCollate. It provides a number of instances of variation in order and variation in structure and shows how they are processed by HyperCollate. In conclusion we reflect upon the implications of such a tool for our definition of "text" and the study thereof, and we argue that the hypergraph allows us to fully deploy XML's potential for textual research.

2. Background

2.1. Context: Computational Philology

The modelling of textual objects and textual collections has become a fundamental feature of computational humanities research (cf. McCarty 2004; Van Hulle 2016; Bleeker 2017). The present discussion takes place against the backdrop of manuscript studies, a field that presents fertile ground for multidisciplinary research. The methodologies and approaches related to this kind of research can be grouped together under the label of "computational philology". The creation of formal models has proven to be a highly useful intellectual tool for textual scholars and philologists who are compelled to express their notions and understandings in a structural, systematic way. At the same time, the complexity of historical texts presents original and complex puzzles for computer and information scientists.

This is certainly true in the case of the modern manuscript dating from early 20th century, which has been described as "complicated web of interwoven and overlapping relationships of elements and structures" (Vanhoutte 2007, "Electronic Textual Editing: Prose Fiction and Modern Manuscripts: Limitations and Possibilities of Text-Encoding for Electronic Editions"). The key question is how to capture, represent, and analyse these features.

The widespread use of XML within the humanities community has given scholars a powerful tool to express their interpretation and knowledge of a text. As indicated above, the representation of a manuscript text in the XML data model facilitates the study, representation, and analysis and, ideally, promotes interoperability and data exchange. Yet the XML data model also presents a number of challenges for the modelling and processing of humanities texts. A complete overview of these issues and proposed solutions is not within the scope of this paper; others have done so elsewhere (e.g. S-McQ & Huitfeldt 2000; De Rose 2004; Piez 2008, 2014).

The present paper focuses on two crucial properties of humanities text. First, text is non-linear but the order in which XML elements occur is significant. Second, its structure often carries implicit semantic meaning that is only partially expressed by an associated schema. If we want to model humanities text in a way that promotes further research, we need to accommodate and anticipate these properties. The non-linearity implies that there exist multiple "paths" through the text; the fact that a structure reflects a certain perspective implies that different texts may have different structures.[1]

The following sections explore to what extend these properties are represented in existing models of text.

2.2. Modelling Properties of Text

One of the most widely used tools for text modelling in the humanities is the TEI Guidelines: a set of extensive recommendations for the transcription and encoding of literary texts created by the TEI (Text Encoding Initiative). Since the TEI Guidelines are developed by the text editing community, they are considered to be the "de facto standard for text encoding" which makes TEI-XML the "lingua franca of current scholarly editions" (Andrews 2013; Pierazzo 2015, 30; Van Zundert 2016, 2015). Although TEI concentrates primarily on functional aspects of text (chapters, paragraphs, named entities, etc), TEI-XML tags can have semantic meaning. For instance, a fragment of textual data wrapped in the XML-TEI element <add> implies that the transcriber considers those sequences of characters as an addition. The exact meaning of the tag, though, may vary depending on the understanding of the transcriber and the goal of the transcription. Markup, then, provides a way to express a view or a perspective on a text; the use of the specific tags and thus the structure of the XML file is clarified in the associated schema.

In order to comply with the diversity of textual models and scholarly concepts, the TEI Guidelines are flexible: they provide a number of TEI schemata and also allow for schema customisation. This particular feature gives scholars a means to express the idiosyncratic characteristics of literary text and to capture a wide variety of textual phenomena by creating their own set of tags. Incidentally,

[1]Additionally, there may be different structures within one text. In XML these additional structures are expressed with milestone elements in order to avoid the well-known issue of overlapping hierarchies. The TAG data model does support this feature, but it is not within the scope of the present article, which focuses on comparing structures between different XML files. See [ref] for an introduction to the TAG data model and how it deals with different challenges for humanities text modelling.

this particular feature also hinders reusability and exchange of data - a complication that may be solved by linking the transcription to existing ontologies (cf. Eide 2009) but that is a different discussion altogether. In short, we can say that, together with the associated schemata, the TEI-XML markup reflects a set of scholarly choices and theoretical concepts related to text. From here on forward, the term "TEI text" is used here when referring to text-centric XML of literary texts.

2.2.1. Multiple Paths

The non-linearity of TEI texts is best illustrated with a case study. Figure 1 shows a fragment from an authorial manuscript from Mary Wollstonecraft Shelley's *Frankenstein* followed by a simplified TEI-XML encoding of the inscriptions on the manuscript.

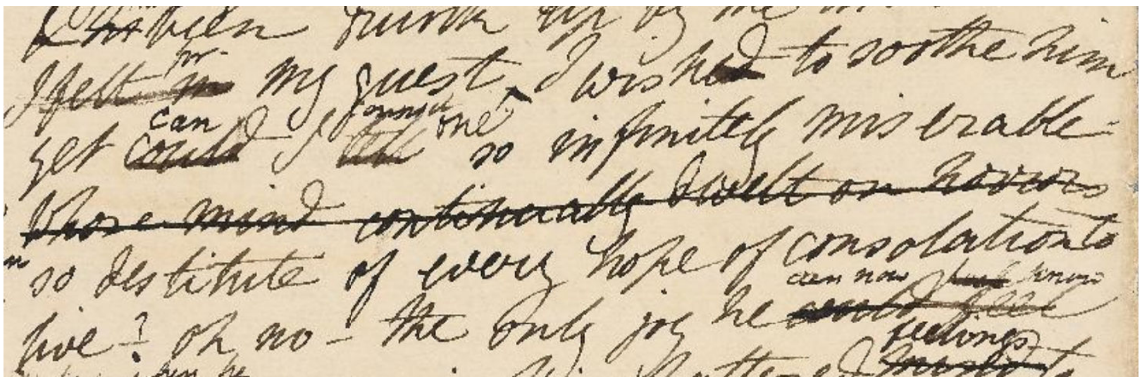


Figure 1. MS. Abinger c.57, fol. 85r (fragment).

```
<TEI>
  <text>
    <s>I wish<del>ed</del> to soothe him
    <lb/>yet <del>could</del><add>can</add>
      I <del>tell</del><add>console</add>
    <add>one</add> so infinitely miserable
    <lb/><del>whose mind continually dwelled on horrors</del>
    <lb/>so destitute of every hope of consolation
      to live <del>-</del><add>?</add>
    oh no - the only joy</s>
  </text>
</TEI>
```

Here, the deleted and added words are tagged with a <del> and <add> element respectively. The parent <s> signifies a sentence and the <lb/>s signal the start of a new line on the manuscript. Together, these markup elements reflect the scholars analysis of the manuscript text and the inline revisions.

Note that the order of the words in the transcription is the result of the scholar's interpretation of the manuscript. To be more precise: the difference between

<del>could</del><add>can</add> and <add>can</add><del>could</del> is meaningful for the transcriber because it implies a chronological order. In some cases, for instance when the author has manually corrected their typescript, the chronological order in which the revisions are made is clear enough. Yet with complex manuscripts this temporal aspect is not always evident and, thus, the transcription reflects a scholarly interpretation.

The manuscript fragment in figure 1 contains what can be called "intradocumentary variation", that is, textual variation *within* one text. Conventionally western scripts are read from left to right, so the linear order in which the characters are placed represents the order of reading. Intradocumentary variation, on the other hand, implies that there are different orders - or "paths" - through the text. In the transcription of figure 1, the first path starts by the <del> element; the second path is indicated by the <add> element. Both paths join after the closing </add> tag. It's important to realise that these paths are located on the same level in the XML tree, so the order in which they are processed by a parser is arbitrary.

In other words, intradocumentary variation, such as additions, corrections, and deletions imply multiple "paths through" or orders of the text. The text is no longer a linear sequence of tokens, but rather a linear graph that is partially ordered: the path through the text splits in two directions and rejoins at a later point.

Plain text, on the other hand, is fully ordered. Hence, as said, the processing of TEI texts as plain-text characters implies that partially ordered data is transformed into fully ordered data. In some cases, the markup can be discarded without much consequence for the order of the text data, for instance in case of named entity tags. With regard to the fragment of figure 1, however, taking out the XML elements and flattening the text data results in a nonsensical sentence:

```
I wished to soothe him yet could can I tell console one so infinitely ▶
miserable
```

If the XML elements including their textual content are removed, we get a similarly illogical result:

```
I wish to soothe him yet I so infinitely
miserable
```

Without the markup we lose the start and end indicators of the different paths, because plain text characters are, by definition, placed in one linear order.

Preparing TEI-XML transcriptions for processing thus necessitates the selection of certain markup elements and the text data they contain. It may be clear that with complex authorial manuscripts containing multiple <del> and <add> elements, like Wollstonecraft's *Frankenstein*, this selection process is not straightforward. Furthermore, structural markup like <s> and <lb/> arguably carries valuable intelligence as well, even if ignoring these elements doesn't immediately impede the flow or legibility of the text.

2.2.2. Structure

The structure of text-centric XML files has been the topic of ongoing research. It is well-known that the textual objects in the XML tree are organised as a single-ordered hierarchy (OHCO; see also Coombs et al. 1987; DeRose et al. 1990; Renear et al. 1996). Some textual features are more naturally represented in XML's tree model than others, but with the aid of workarounds it is perfectly possible for scholars to encode all information about intradocumentary variation that is relative or important according to the conceptual model they follow. Proposing an alternative to XML, therefore, would bypass the widespread use of XML in the humanities community.

However, it is worthwhile to take a closer look at the way TEI texts currently model textual structures and what that implies for processing. It may be clear that a document's structure reflects a specific perspective or view on the text. That is, the structure of a text that is studied from a material or bibliographic perspective will most likely be different than the structure of the same text when examined from a temporal perspective. The different perspectives are reflected in the selection of elements as well as the tagging of text data. In the monohierarchical data model of XML, a TEI text contains but one structure. Any additional structures are represented with <milestone>s or empty elements.

2.3. Analysis Through Collation

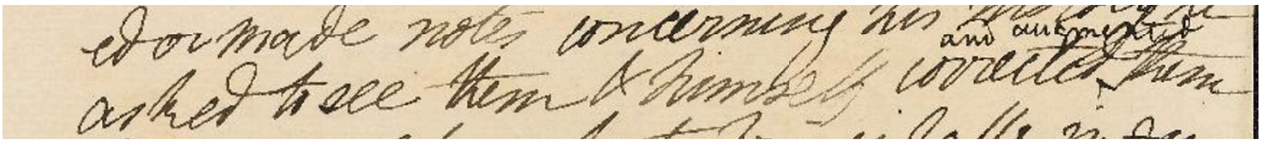
TEI texts can be analysed in many ways and for many different research purposes. This paper focuses on a widely used scholarly primitive to analyse text: collation. Collation at its most basic level means the comparison of two or more texts (literally "placing side by side"). The outcome of this comparison presents an overview of the variance between a number of text versions ("witnesses"). In general, texts are collated for three different reasons:

- to track the transmission of a text;
- to get as close as possible to the original text;
- to establish a critical or final text. This could also mean the generation of a list of variants and/or a critical apparatus.

Comparing a high number of witnesses is a tiresome and error-prone activity for humans, making it an ideal candidate for automation. To this end, several collation tools have been developed over the past decades. It is not within the scope of this paper to present a full discussion of the different methods these tools employ, but it suffices to say that they focus on comparing strings of plain text and therefore do not address the characteristics of TEI text as discussed above. [2]The consequences of transforming TEI text to plain text have been outlined in sections 2.2.1 and 2.2.2 already; the following section takes a closer look at the implications of that processing method.

2.3.1. Collation and Multiple Paths

Figure 2 shows again a fragment from the authorial manuscripts of *Frankenstein*; figure 3 shows a different version of the same sentence. A simple TEI-XML transcription is presented below the respective fragments.



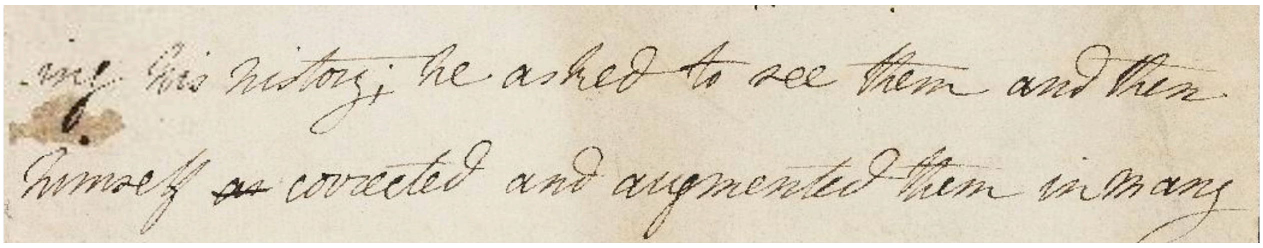
edor made notes concerning his [illegible] and augmented
asked to see them & himself corrected them

Figure 2. Witness 1 (fragment of MS. Abinger c.57, fol. 85r).

Transcription Witness 1:

```
<TEI>
    <p>
        <s>& himself corrected <add>and augmented</add> them</s>
    </p>
</TEI>
```

In Witness 1 the textual data has one parent <s> and the addition is encoded with an <add> element. The <add> element, then, represents the start of intradocumentary variation, i.e. the diversion of two paths.



ing his history; he asked to see them and then
himself ~~as~~ corrected and augmented them in many

Figure 3. Witness 2 (fragment of MS. Abinger c.58, fol. 22r).

Transcription Witness 2:

```
<TEI>
    <p>
```

[2]See Bleeker 2017 for a more extensive discussion of existing approaches to automated collation. Furthermore, a short overview of historic approaches to collation is also available in a Jupyter Notebook, created for the DiXiT workshop 'Code and Collation'. The workshop material is available as Jupyter Notebook, see http://nbviewer.jupyter.org/github/DiXiT-eu/collatex-tutorial/blob/master/INTRO.ipynb

```
            <s>and then himself corrected and augmented them</s>
        </p>
    </TEI>
```

Witness 2 has a similar structure, but the addition is now part of the running text. In other words, there is but one path through the text of this witness. Together, these witnesses illustrate that variation within one text can be related to other text versions.

However, a conventional collation program, that focuses on plain text data only, discards all information about the multiple paths in Witness 1:

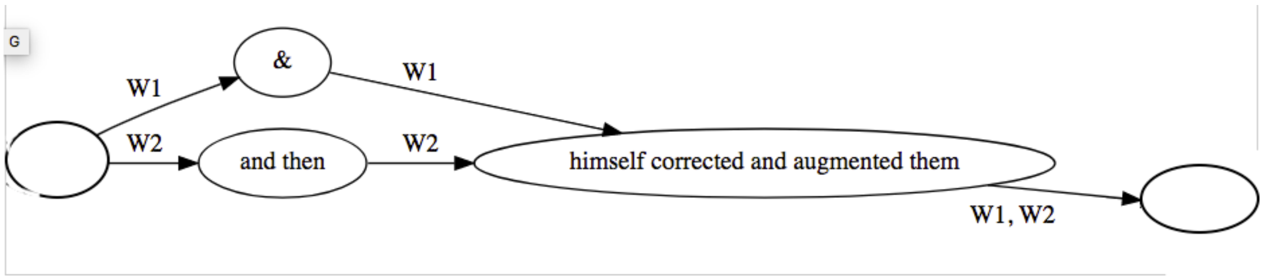


Figure 4. Output of collation between Witness 1 en Witness 2, rendered as variant graph

The collation result is visualised as a variant graph, a commonly used data structure to store and represent textual variation. Like any graph, a variant graph consists of nodes and edges. The nodes represent textual content and the edges are directed and represent the order in which the text should be read (i.e. following the directed edges, from left to right or from top to bottom in a vertical representation). Every edge contains a label, a so-called "siglum", that refers to the witness or witnesses. Here, the siglum "W1" refers to Witness 1 and "W2" refers to Witness 2. By following a siglum through the variant graph, the reader can read the text of the associated witness. The text data is segmented based on the transition between alignment and variation: aligned text is placed in the same node.

The collation result represented in figure 4 does not include the temporal aspect of the writing process that is represented by the addition in Witness 1. As a consequence, the result doesn't communicate that literary text develops in stages, nor does it convey how the stages of one witness can be related to other witnesses.

The current automated collation tools are schema-independent: they collate character strings and necessitate a selective approach where only (parts of) textual data is collated. In order to compare witnesses that each have multiple paths through the text, the collation tool needs to understand and recognise markup elements that indicate the start and the end of the path. To take the example above, the tool needs to know that the <add> tag represents the start of a path, and that the </add> tag represents the end of a path and thus the end of the intra-documentary variation. This implies that the tool must be "schema-aware".

2.3.2. Collation and Structure

The way regular automated collation tools deal with the structure of TEI texts is best represented by a new example. Figure 5 and figure 6 present two manuscript fragments that contain have more or less the same text but with a different structure.

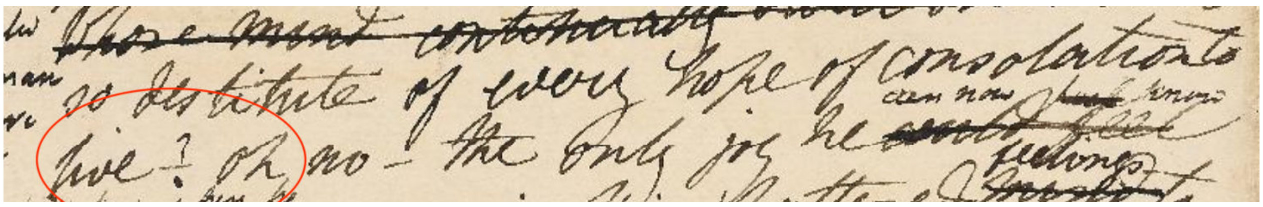


Figure 5. Witness 1 (fragment of MS. Abinger c.57, fol. 85r).

A relatively simple example already reveals that a particularly small revision results in a structural change:

The sentence first reads "...so destitute of every hope of consolation to live - oh no - the only joy...". A simplified XML-TEI encoding of the sentences would look as follows:

```
<TEI>
    <text>
        <s>so destitute of every hope of consolation to live - oh no - ▶
the only joy</s>
    </text>
</TEI>
```

in which the textual data has one parent <s/>. Another witness contains more or less the same text, but split up in two separate sentences (figure 6):

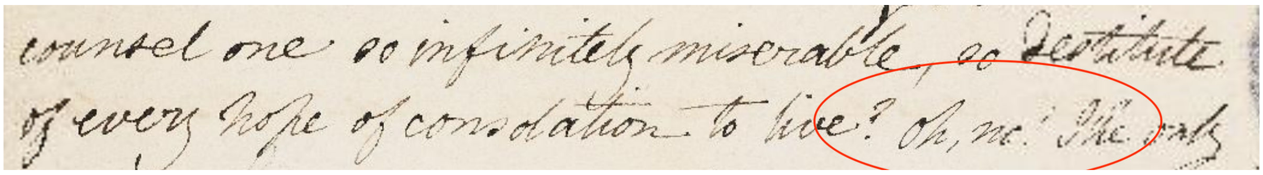


Figure 6. Witness 2 (fragment of MS. Abinger c.58, fol. 22r).

which results in the following TEI-XML encoding:

```
<TEI>
    <text>
        <s>so destitute of every hope of consolation to live?</s> <s>Oh ▶
no!</s>
    </text>
</TEI>
```

Apart from the change to capitals and some punctuation marks the textual data has remained the same, but is now divided over two parent <s> elements. A standard collation tool that looks at text data only, would not spot this difference in structure:

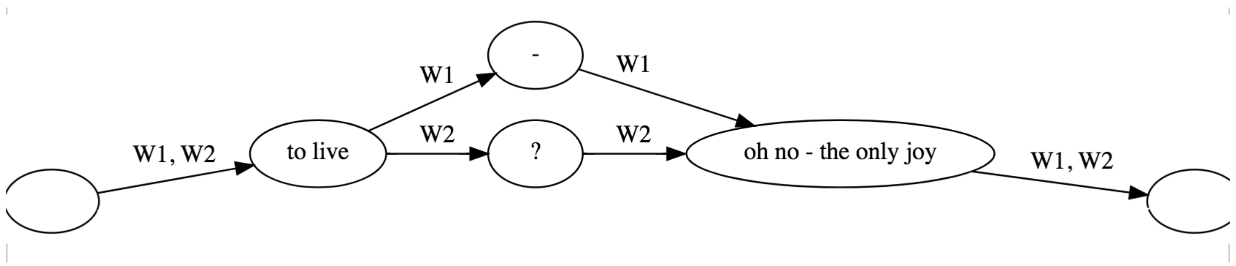


Figure 7. Output of collation of Witness 1 and Witness 2, rendered as a variant graph

It is, of course, also possible that a word changed at the same time as the markup changed. Simply prioritising structure over text data would therefore not result in the desired outcome either. For that reason, we created a new algorithm that processes TEI texts with multiple paths and compares the witnesses on a structural as well as a textual level. Using a hypergraph data model, HyperCollate is able to process TEI texts in an inclusive manner that significantly improves the collation result.

3. Approach

3.1. Context

In order to better understand the approach of HyperCollate, it is worthwhile to briefly explore how it relates to existing collation algorithms. We take the open source collation tool CollateX as point of departure.[3]

Section 2.3 already mentioned that, in general, collation tools strip the TEI text of its markup. After converting the XML file into plain text, CollateX subsequently tokenises the string of characters on whitespace and punctuation, and collates the strings using the global alignment algorithm of Needleman-Wunsch. Needleman-Wunsch is a well-known algorithm to align two sequences of objects against each other.[4]In the case of TEI text, these objects are characters or tokens. Each sequence is linear in nature, this means that at every position in the sequence there is only one object. Now the goal of alignment is to find the smallest amount of changes to turn one sequence into the other.

The collation result is then stored internally as a variant graph. It can visualised in different ways, depending on the preference of the user. The strengths of

[3]See https://collatex.net/docs
[4]See Needleman and Wunsch 1970 and https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm

the variation graph are focus and simplicity: once users understand how it should be read it is relatively easy to present variation between witnesses. A disadvantage of the variant graph is that it only stores plain text and variance *between* witnesses: additions and deletions within a single witness, and structural differences between witnesses are are ignored (see section 2.3). Note that a variant graph is partially ordered: places of variation, on the same rank, do not have an order. This is different from the tree model behind OHCO that is, as said, fully ordered.

In the case of more than two witnesses, the current version of CollateX makes use of "progressive alignment", meaning it doesn't compare all witnesses at the same time. Instead, it first compares two witnesses, stores the result of that comparison in a so-called variant graph and then progressively compares another witness against that graph, at every turn merging the result of that comparison into the graph until all witnesses are merged.

As mentioned in section 2, it is clear that dealing with partially ordered XML poses a number of challenges. In short, it is a challenging combination of fully ordered text data with a structure that, at the points in the text where variation occurs, is unordered. As a consequence, there exist multiple "paths" through the text that all need to be taken into account during the collation process. Furthermore, the fact that a structure reflects a certain perspective implies that different texts may have different structures, which needs to be addressed as well.

3.2. modelling (HG)

HyperCollate uses a directed property hypergraph data model to store the witness data and the result of the collation. All graphs consist of nodes and edges. The nodes represent objects and the edges connect the nodes to one another. In commonly used graph models, such as a directed acyclic graph (DAG), graph edges connect one node to one another node in the graph. In a hypergraph, conversely, edges are hyperedges. Hyperedges connect one or more nodes with each other. Standard edges are undirected, that is, the relation that an edge between two nodes applies in both directions. If there is an edge between node A and node B, A is associated with B, as well as B with A. Directed hyperedges contain an order. A directed hyperedge points from one node to another. A directed hyperedge connects one node, "the head", to multiple (one or more) nodes, the "tail". A property graph is a kind of graph were extra information in the form of properties can be stored on the nodes and edges. For example, every node has a type property.

The hypergraph model for textual variation consists of the following node types:

- Text nodes; a single text node contains a piece of textual content. All the text nodes together represent the full text of the set of witnesses.

- Markup nodes; the markup nodes contain all the information associated with the markup; such as the tag name and the attributes contained in the original XML TEI files.
- EmptyText node; these are text nodes with empty content, meant as targets for markup nodes that represent milestone elements from the original XML TEI files.
- Start and end nodes; these are a special kind of EmptyText nodes; they denote the beginning and end of the text and facilitate traversing the hypergraph and allow for variation that occurs at the beginning and end of witnesses to be recorded.

The hypergraph model for textual variation consists of the following edge types:

- Text to text directed edges; these connect one text node to another and in that manner store the order of the text. Note that one text node can have multiple outgoing edges. Multiple outgoing edges are used in the case of textual variation. See for more information section 3.1.2 The edges contain sigli information as a property to indicate with witnesses they are associated.
- Markup to text directed hyperedges; these connect one markup node to one or more text nodes. The hyperedges are labeled with the sigli of the witnesses involved as a property.

3.2.1. Multiple Paths

The hypergraph for variation is an evolved model based on the variant graph (see section 2.3). To tackle the first requirement, i.e. processing texts with multiple paths, we release the constraint that at every point of variation in the graph every witness may occur only once. Now, in case of intradocumentary variation in witness N, the siglum N will occur multiple times at the same rank in the graph. It is import to note, that just like in the case of intradocumentary variation, there is no ordering in the variation at the same rank. We do, however, keep track of whether each of the variants is an addition or a deletion. This information can be used e.g. for visualisation purposes.

An empty variant hypergraph consists of only one start node and one end node, connected by a directed edge from the start node to the end node.

In a variant hypergraph text the textual content is segmented into text nodes, the nodes are as large as possible without a change in markup occurring between two consecutive text nodes. A simple text without markup is thus recorded in three nodes: one start node, one text node containing the textual content, and one end node. A directed edge connects the start node to the text node. Another directed edge connects the text node to the end node.

In order to encode textual variation, the textual content is segmented into several nodes. The segmentation takes place at the transition between aligned pieces of text and variation, and vice versa. Each text node has one or more incoming and outgoing directed edge. Every edge has a property on it indicating which witness this edge belongs to. It is thus possible to traverse the hypergraph starting at the start node, walking over the text nodes and following the edges labeled with the witness siglum to the end node. For every witness there is always a fully connected path through the hypergraph from the start node to the end node. Textual variation is stored as a node at the beginning of the variation with multiple outgoing directed edges. In the case of variation between witnesses, the sigli on the outgoing edges have to be unique and a siglum can not be repeated. In the case of intradocumentary variation, the same siglum will be repeated.

3.2.2. Structure

The second requirement is to store markup in the data model. Similar to the OHCO model, markup is stored in its own nodes. We added a new type of node, specifically aimed at markup, in addition to the text node types. The markup nodes store the tag name and the attributes involved. Note that, just like in OHCO, for every combination of open and end tags only one markup node is added to the hypergraph. Now the text nodes and the markup nodes are connected to one and other through the use of directed hyperedges. The edges point from the markup node to the text nodes. Just like in the variant graph every hyperedge has sigli attached to it.

In contrast to the variant graph, text nodes are not only segmented based on textual variation/non variation. In the variant hypergraph model the segmentation of the text nodes is also based on markup variation. The segments contain as large a piece of text as possible, within the constrains that there may be no change in markup or textual content or punctuation between witnesses for that segment.

3.3. Analysis through collation: HyperCollate

The nature of TEI texts entails that the textual content should either be treated equal to the structure of the file, or that it be given priority. In previous experiments we found that only giving priority to text data did not always give satisfactory results. In TEI texts containing a significant amount of markup, collation results improved when text and markup were considered as equally important. Furthermore the current automated collation tools are schema-independent: they collate character strings and necessitate a selective approach: only (parts of) textual data is collated. If the collation tool needs to take the markup into account for the collation, it needs to understand and recognise markup elements. In other words: it needs to be schema-aware.

HyperCollate operates in four main steps:

1. Converting the XML tree of all of the witnesses into a hypergraph for each one.
2. Align two hypergraphs
3. Merge the two hypergraphs into one.
4. Repeat steps 2 and 3 for the remaining witnesses in case of more than 2 witnesses.
5. Visualise or export the resulting hypergraph.

There are two reasons for HyperCollate's first main step: converting the TEI XML witnesses into the hypergraph model before the alignment. Firstly, converting all the witnesses to graphs before alignment makes the alignment process uniform, especially for progressive alignment. For progressive alignment, or collating more than two witnesses, we take the result of the previous collation, which is a graph, and collate the next witness into it. If we would not convert the first two witnesses to a graph structure before alignment, the alignment problem would be different for the first two witnesses as opposed to the rest: the first two witnesses would present us with a tree to tree alignment problem, while the remaining witnesses would pose graph to tree collation problems.

Secondly, and perhaps most importantly, the variant hypergraph represents textual variation in a more accurate and explicit manner. After the conversion process every markup node in the TEI XML source is represented by one markup node in the resulting variant hypergraph. In an XML tree structure the markup elements are on top level, the text elements are at the bottom (so-called "leaf nodes"). In the hypergraph model for variation, however, the text elements are at the centre of the model and the relationship between text nodes and markup nodes is expressed by hyperedges. Now that every witness we want to collate is converted into a hypergraph, the witnesses can be aligned (step 2 of HyperCollate's operation process). The goal is to align the nodes of the two hypergraphs in such a way that we find the smallest possible amount of changes needed to change one hypergraph into the other: we want to find the largest amount of nodes that are "aligned".

The HyperCollate algorithm is a global alignment algorithm that is inspired by Needleman-Wunsch. It extends the Needleman-Wunsch algorithm in three ways:

- Instead of aligning two sequences of tokens as before, HyperCollate aligns two sequences of matches. Due to this, HyperCollate uses four edit operations (skip next match of sequence 1, skip next match of sequence 2, align next match of sequence 1, align next match of sequence 2) instead of three (add/remove the next token from sequence 1, add/remove the next token from sequence 2, align/replace the next token sequence 1 and 2). By aligning matches rather than tokens we deal with the fact that the variant hypergraph is partially ordered;

- Whereas Needleman-Wunsch uses one type of object in the sequence, HyperCollate uses two object types: one for text and one for markup tags open and close events. By having two types of matches (one for markup; one for text) we treat text and markup with equal priority during the alignment;
- Finally, after the initial alignment pass which aligns all content, HyperCollate performs a second alignment pass where it distinguishes between changes in markup and changes in text. This refinement pass improves the alignment quality by aligning the text of both witnesses separately from aligning the markup of both witnesses.

Based on the aligned nodes we can merge the two witness hypergraphs into a new hypergraph that contains the same information. All the nodes that are not aligned are unique to one of the two hypergraphs; all the nodes that are aligned can be reduced from two to one, with labels on the edges indicating which node is part of which hypergraph. The resulting merged hypergraph can be visualised or exported.

Going back to the example mentioned in section 2.3.2 provides us with the following two witnesses, simplified for reasons of brevity and clarity:

```
<TEI>
    <p>
        <s>and then himself corrected and augmented them</s>
    </p>
</TEI>
```

```
<TEI>
    <p>
        <s>& himself corrected <add>and augmented</add> them</s>
    </p>
</TEI>
```

Using HyperCollate to align and merge these witnesses results in a variant hypergraph which can be visualised as follows:

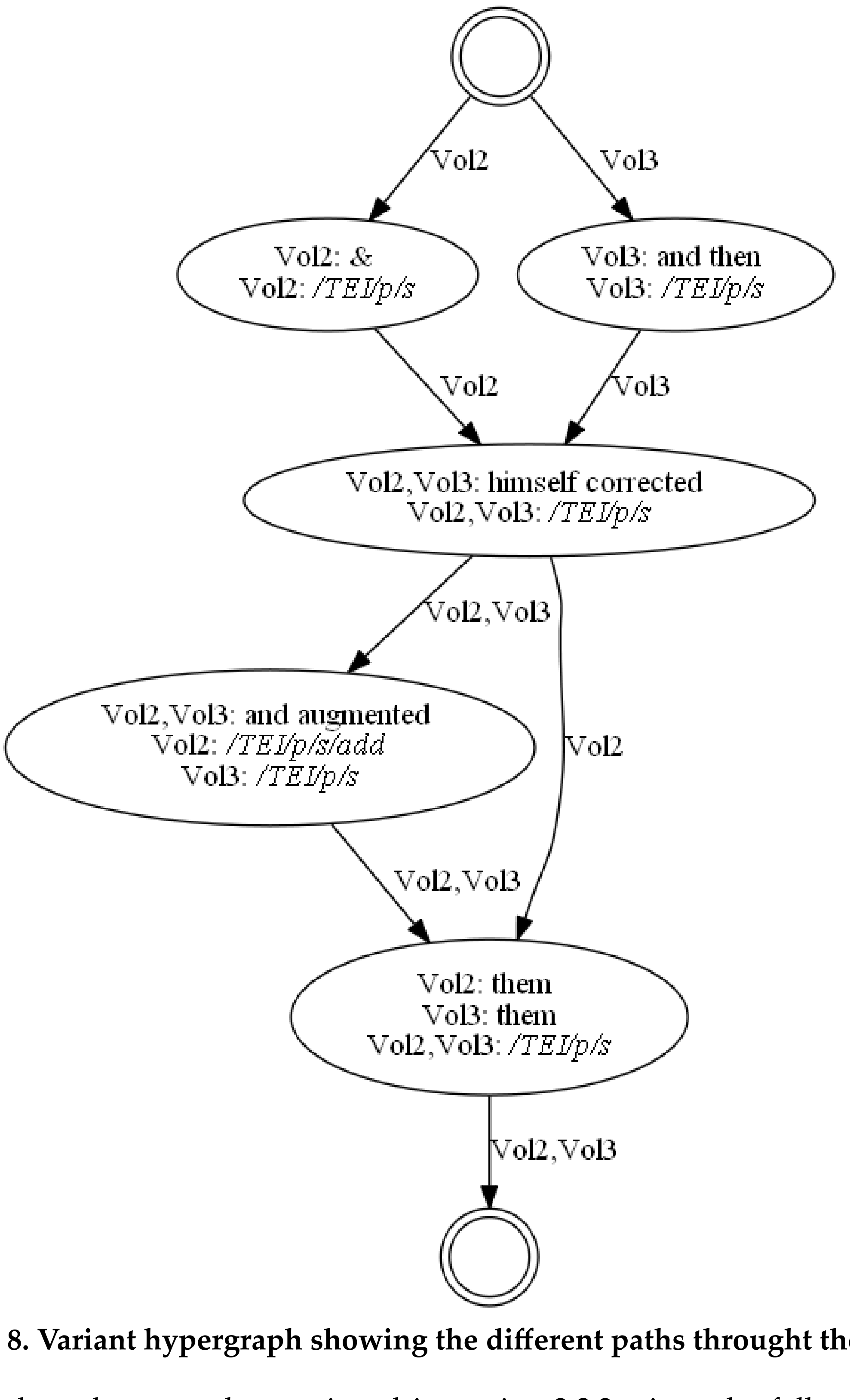


Figure 8. Variant hypergraph showing the different paths throught the text.

Going back to the example mentioned in section 2.3.2, given the following two witnesses, simplified for reasons of brevity and clarity:

Witness Vol2 (fragment fol. 85r):

```
<TEI>
    <text>
        <s>to live - oh no - the only joy</s>
    </text>
</TEI>
```

Witness Vol3 (fragment fol. 22r)

```
<TEI>
    <text>
        <s>to live?</s> <s>Oh no - the only joy</s>
    </text>
</TEI>
```

the resulting variant hypergraph looks like this:

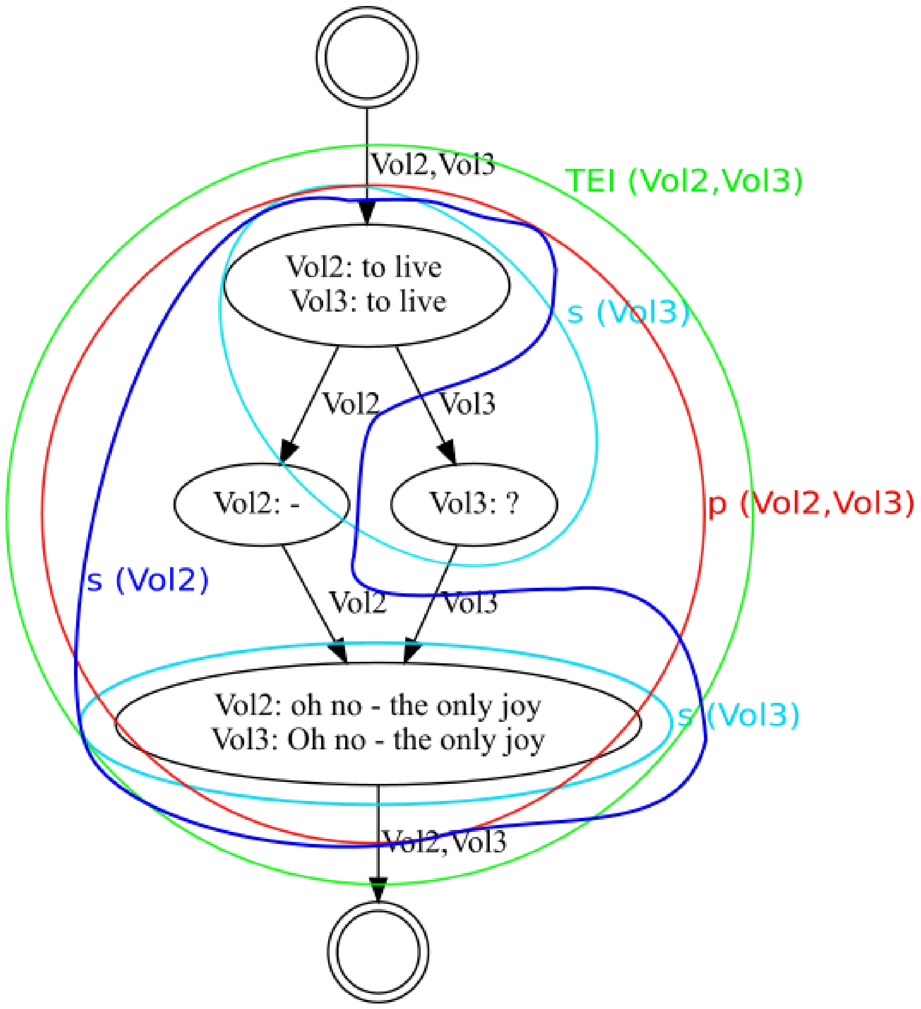


Figure 9. Variant hypergraph showing the difference in structure between the two witnesses.

4. Conclusion

XML is a key instrument in the modelling and processing of texts for the humanities. The encoding of a text is influenced by the perspective of the scholar, ensuring that TEI-XML encoded texts contain the scholar's critical analysis of the text, its revision, its modes of production, etc. This kind of TEI-XML file we define as text-centric XML of literary texts: partially ordered data files that, due to their dual nature, pose a number of challenges for traversing, parsing, and processing. We argue that by using a hypergraph data structure it is possible to model and process TEI texts. Furthermore, we demonstrate how the hypergraph can be used to model partially ordered data, in particular by describing how it processes texts that contain multiple paths and variation between structures. This entails a departure from the prevailing idea of processing text as a linear sequence or a tree.

The hypergraph data structure supports different forms of document modelling and processing, but we focus on the modelling of textual variation. Collation is a frequently used method to analyse text and textual variation, and forms an exemplary vehicle to illustrate the implementation of a hypergraph model for text. We present HyperCollate, a collation tool that uses the TEI-XML markup to produce a collation result that accurately reflects the scholarly knowledge and understanding of text. HyperCollate accepts (TEI-encoded) XML files and uses information that is stored in these files to significantly improve the result of the analysis. Partially ordered data is especially challenging for algorithms, because an algorithm cannot simply either ignore or apply order everywhere, but needs to decide on a case to case basis whether the order is important or not. HyperCollate uses domain-specific knowledge to handle this problem. It results in an exhaustive representation of the variance within and between different versions of a literary work, thus allowing scholars to better analyse the dynamic nature of text. Consequently, the approach makes optimal use of the potential of the XML data model for textual research.

Multi-Layer Content Modelling to the Rescue

Erik Siegel
<erik@xatapult.nl>

1. Summary

There are companies, for instance educational publishers, that have lots of content for a multitude of different products. These products share some content aspects but unfortunately also differ wildly in things like structure, metadata and mark-up. From an information modelling perspective, a different model would be required for each of them.

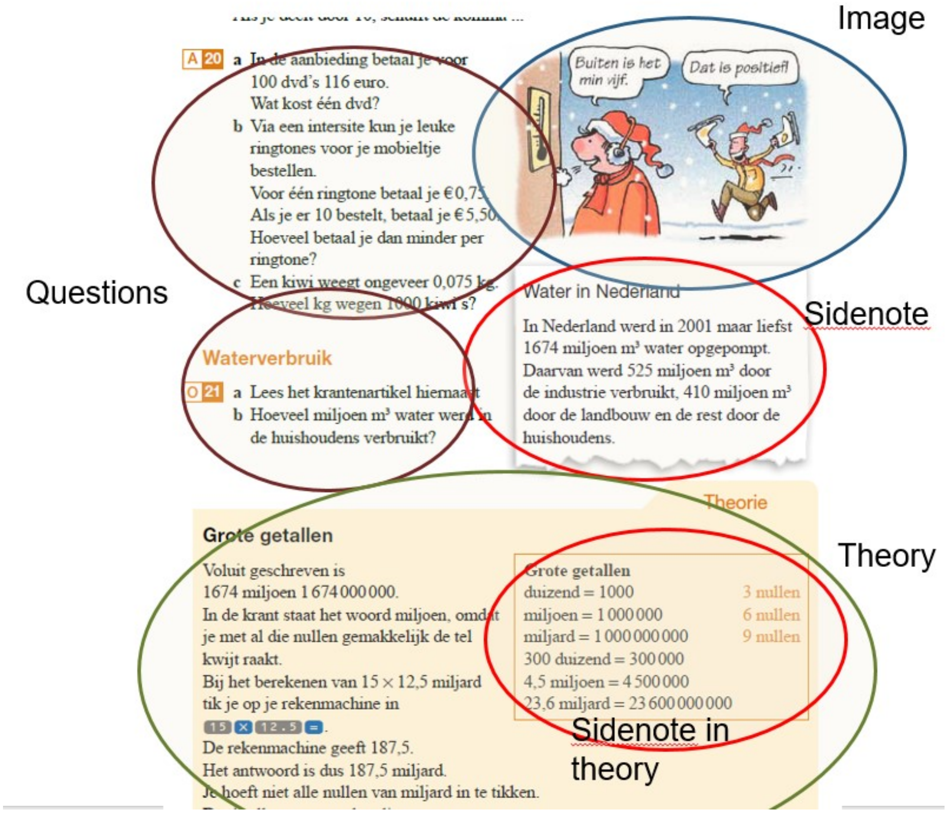
Different models would mean different schemas, authoring tool configurations and publishing tool-chains. A huge investment and hard to maintain. A way to circumvent this is by introducing *multi-layer content modelling*:

- You start off with a base content model that is flexible enough to cater for the needs of all the products. This could be a standard like Docbook or DITA or something bespoke.
- On top of that you add a second layer of modelling to cater for the product differences. This layer must be able to describe and validate the product specific contents, but also guide authoring tools and other software in doing the right, product specific, thing.

2. Setting the stage

Most of my customers are educational publishers and they have a content modelling problem. All of them have many product families which are similar in nature but very different in details. Let's have a look at some examples.

Here is a page from a secondary education book. The publisher made it look attractive and playful, but we, as XML geeks, can see the underlying structure.



Als je deelt door 10, schuift de komma ...

A 20 a In de aanbieding betaal je voor 100 dvd's 116 euro. Wat kost één dvd?
b Via een intersite kun je leuke ringtones voor je mobieltje bestellen. Voor één ringtone betaal je € 0,75. Als je er 10 bestelt, betaal je € 5,50. Hoeveel betaal je dan minder per ringtone?
c Een kiwi weegt ongeveer 0,075 kg. Hoeveel kg wegen 1000 kiwi's?

Waterverbruik

O 21 a Lees het krantenartikel hiernaast.
b Hoeveel miljoen m³ water werd in de huishoudens verbruikt?

Water in Nederland

In Nederland werd in 2001 maar liefst 1674 miljoen m³ water opgepompt. Daarvan werd 525 miljoen m³ door de industrie verbruikt, 410 miljoen m³ door de landbouw en de rest door de huishoudens.

Theorie

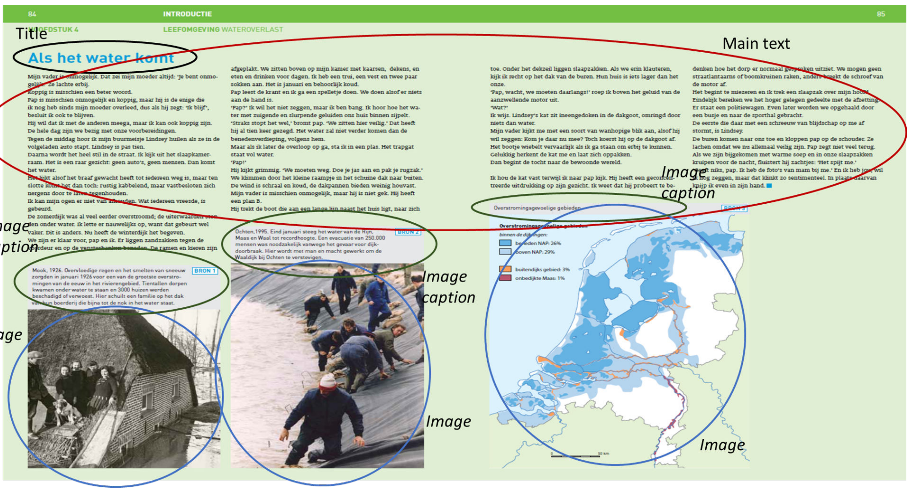
Grote getallen

Voluit geschreven is 1674 miljoen 1 674 000 000.
In de krant staat het woord miljoen, omdat je met al die nullen gemakkelijk de tel kwijt raakt.
Bij het berekenen van 15 × 12,5 miljard tik je op je rekenmachine in
15 × 12.5 =.
De rekenmachine geeft 187,5.
Het antwoord is dus 187,5 miljard.
Je hoeft niet alle nullen van miljard in te tikken.

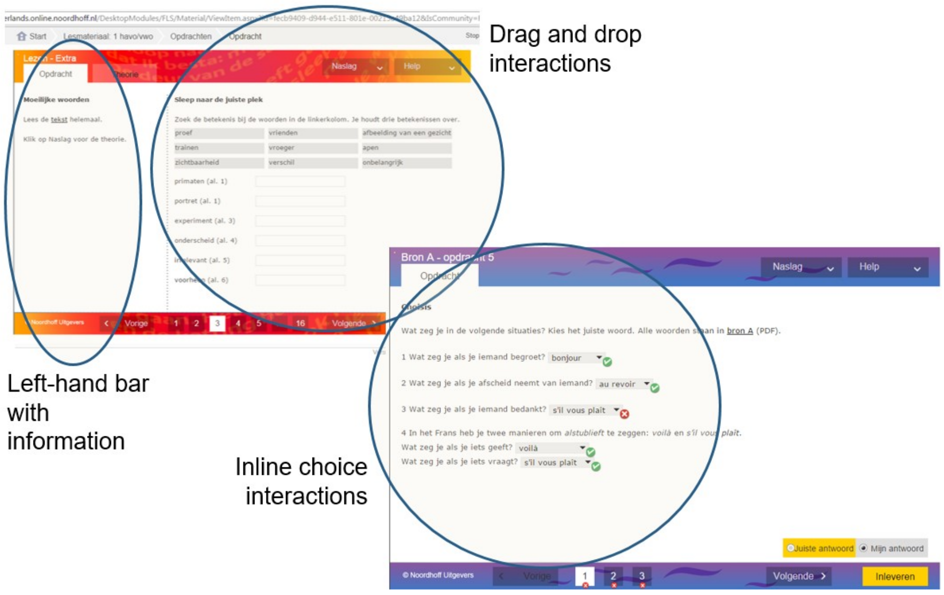
Grote getallen	
duizend = 1000	3 nullen
miljoen = 1 000 000	6 nullen
miljard = 1 000 000 000	9 nullen
300 duizend = 300 000	
4,5 miljoen = 4 500 000	
23,6 miljard = 23 600 000 000	

Of course this must not only be published as a book. Multi-channel publishing is a must, so we will have to be able to translate this to the screen and make the questions truly interactive.

Now have a look at this, same publishing company, different product:



And here are some screens from an online application:



Moeilijke woorden

Lees de tekst helemaal.

Klik op Naslag voor de theorie.

Sleep naar de juiste plek

Zoek de betekenis bij de woorden in de linkerkolom. Je houdt drie betekenissen over.

proef	vrienden	afbeelding van een gezicht
trainen	vroeger	apen
zichtbaarheid	verschil	onbelangrijk

primaten (al. 1)
portret (al. 1)
experiment (al. 3)
onderscheid (al. 4)
irrelevant (al. 5)
voorheen (al. 6)

Wat zeg je in de volgende situaties? Kies het juiste woord. Alle woorden staan in bron A (PDF).

1 Wat zeg je als je iemand begroet? bonjour ✓

2 Wat zeg je als je afscheid neemt van iemand? au revoir ✓

3 Wat zeg je als je iemand bedankt? s'il vous plaît ✗

4 In het Frans heb je twee manieren om *alstublieft* te zeggen: *voilà* en *s'il vous plaît*.
Wat zeg je als je iets geeft? voilà ✓
Wat zeg je als je iets vraagt? s'il vous plaît ✓

An additional problem is that the volume of content an educational publisher has to produce and maintain is high. To cover everything for the Dutch secondary education school-system, a publisher would need to publish roughly 450 *different* books (over 60,000 pages!) accompanied by approximately 150 different (sub)websites with support material and exercises.

Therefore, a suitable content solution would have to deal with:

- High volume
- Different product families with different structural models and varying markup demands
- Complex interactions and other data structures
- Multiple output channels (print, web, e books, PDF, etc.)

However, solutions with different models for different product families are way too expensive. The total costs for maintaining every model, configuring editors and other tool-chain components will go through the roof. The situation will probably become unmanageable very soon. You'll also end up with content in silos. Not what you want.

3. Multi-layer content modelling

A possible solution for the situation described is to *layer* the content model:

- You start off with a base content-model that is flexible and wide enough for all your content. This could be something existing like DITA or Docbook or something bespoke.

  For this base content-model you can use all the tricks in the XML book to make sure your content is conformant: DTDs, Schemas, Schematron, etc.

- On top of that you layer another content-model that limits the possibilities of the first layer to exactly what is needed for a certain product family.

  For this you don't use the standard XML validation tool-set but create a definition in what we might call a "Domain Specific Validation Language" or DSVL.

This approach will probably sound familiar to you. Layering validation is not an original idea. Here are some examples where this is done also:

- DITA specializations
- RelaxNG customizations
- Schematron in general
- Epischemas (described in a talk by Gerrit Imsieke[1] at XML Prague 2017)

Now these are all technically based languages. They say something about the XML structure in terms of … XML structure (elements, attributes, etc.). What we wanted was a much more functional and domain driven approach to make the creation of definitions as easy as possible.

4. Implementation

To make this a bit more concrete, let's look at an actual implementation of this idea that I did for one of my customers, Infinitas Learning, a group of educational publishers with companies in Belgium, Sweden and The Netherlands.

4.1. The base content model

The content model used is a bespoke one. It shamelessly steals ideas from other standards like DITA, TEI and QTI. For instance maps for holding smaller pieces of content together. It also incorporates complete sub-standards, like MathML for equations. The model is supported by an extensive schema, Schematron rules, examples and written documentation.

Creating a bespoke content model will probably raise some eyebrows but we had good functional and technical reasons for it. One of them was the introduction of the second validation layer, the model was designed with this in mind.

There are a number of aspects of this content model I would like to use for illustration purposes:

- Structure: Instead of this:

```
<chapter>
  …
  <subsection1>
      …
```

[1] http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#d6e4133

```
      <subsection2>
      …
      </subsection2>
    </subsection1>
  </chapter>
```

We do this:

```
  <section class="chapter">
    …
    <section class="subsection1">
      …
      <section class="subsection2">
      …
      </section>
    <section>
  </section>
```

- Markup: Instead of this:

  ```
  <h1>A <i>header</i> that is <b>bold</b></h1>
  ```

  We do this:

  ```
  <p clas="header1">A <em class="italic">header</em> that is
      <em class="bold">bold</em></p>
  ```

- Metadata: Metadata properties are modelled as name/value pairs. Properties are always bundled in metadata-sets. The content-model is orthogonal towards metadata: You can bind a metadata-set to any element in the contents.

  So, for instance, assume there is a metadata-set for interactions called `interaction-info`, with properties like `nr-of-retries`, `score`, etc. You can bind an instance of this metadata-set to an actual interaction in your content. That would look like this:

  ```
  <multiple-choice-interaction id="mc-1">
    …
  </multiple-choice-interaction>

  <metadata-set about-idrefs="mc-1" class="interaction-info">
    <properties>
      <property property-name="nr-of-retries">
        <value>2</value>
      </property>
      <property property-name="score">
        <value>6.5</value>
      </property>
    </properties>
  </metadata-set>
  ```

4.2. Second layer validation

Back to the multi-layer content-modelling. If we split the validation in two layers, what would be done on each of them?

Table 1. Examples of two-layered validation

What	Base layer	Second layer
Structure	• Any section structure to any depth • Any class	• Define the section classes that can be used. • Define how sections with these classes can be nested. • Limit the number of occurrences of sections with a certain class.
Markup	• Limited number of elements • Any class	• Depending on where you are in the document, limit class usage and nesting
Metadata	• Metadata-sets can be attached to elements in the contents • Any property, any value type	• Define what is in a specific metadata-set (properties, value types, nr. of occurrences, etc.). • Define to which elements you can bind a specific metadata-set.

For the implementation of the second layer validation, a so-called DSVL (Domain Specific Validation Language) was created with the following properties:

- It is *not* a generic validation language. The syntax of the DSVL is based on the base layer syntax. For the DSVL to function properly, the content must be valid to the base layer first.
- The DSVL serves many masters:
  - Of course validation of the XML
  - Configuration of the editor, which adapts itself to the definitions in the DSVL

  - The publication tool-chain that looks in the DSVL to find out what it must do with a particular piece of content
  - The content specialist. Creation of a DSVL definition should be relatively easy and straightforward.
- It has limited functionality. To avoid an overly complex DSVL language, we abstained from trying to define every little nitty-gritty detail.

As an example, let's express the examples from the previous section in our DSVL:

- Structure: what we would like to express are things like class-names, nesting, occurrences, etc. In our implementation:

```
<section-definition class="chapter" max-occurs="10">
  …
  <section-definition class="subsection1">
    …
    <section-definition class="subsection2" min-occurs="0">
    …
    </section-definition>
  </section-definition>
</section-definition>
```

- Markup: The DSVL must define allowed class-names. We also had the requirement to be able to limit the number of words in an element. In our implementation:

```
<element-definition element-names="p" class="header1" word-count-
max="15"/>
<element-definition element-names="em" class="italic"/>
<element-definition element-names="em" class="bold"/>
```

- Metadata: for the definition of a metadata-set we would like to know its class-name, the properties it contains, the property data types, etc. In our implementation:

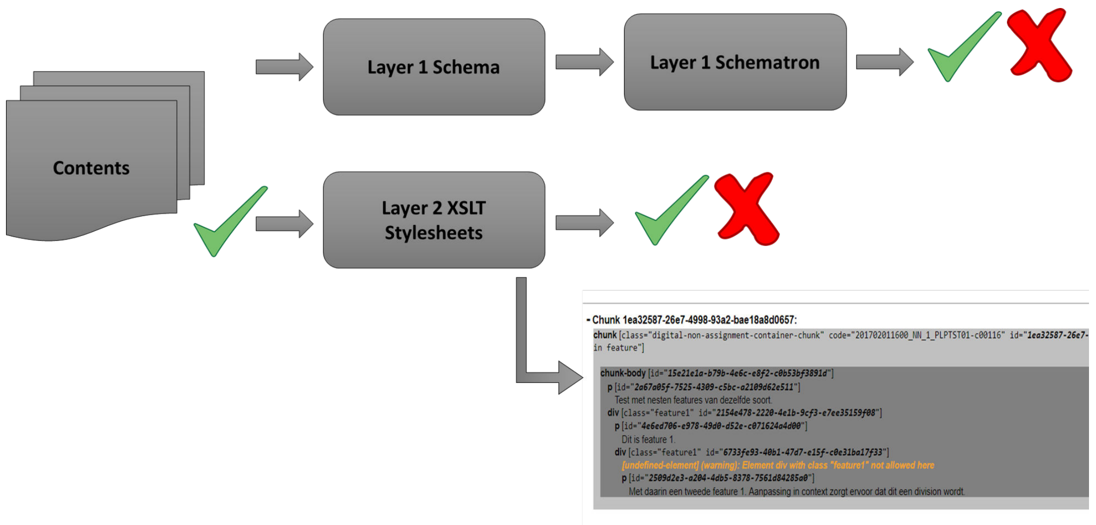
```
<metadata-set-definition class="interaction-info">
  <properties-definition>
    <property-definition property-name="nr-of-retries">
      <value-definition datatype="integer"/>
    </property-definition>
    <property-definition property-name="score">
      <value-definition datatype="double"/>
    </property-definition>
  </properties-definition>
</metadata-set-definition>
```

Now this is not about the choices we made for this XML. Illustrated here is the nature of this “Domain Specific Validation Language”. It does not define element structures or attributes in general. Instead it defines content-structures on a very specific and targeted semantic level: How does the structure look like, what metadata is to be used. It's about the content for a specific product family in much more familiar terms than an XML Schema does.

5. Usage

Now having such a DSVL is fine, but if you can't use it, it's of no help at all. So what did we do with it?

- We used it, of course, for validation:



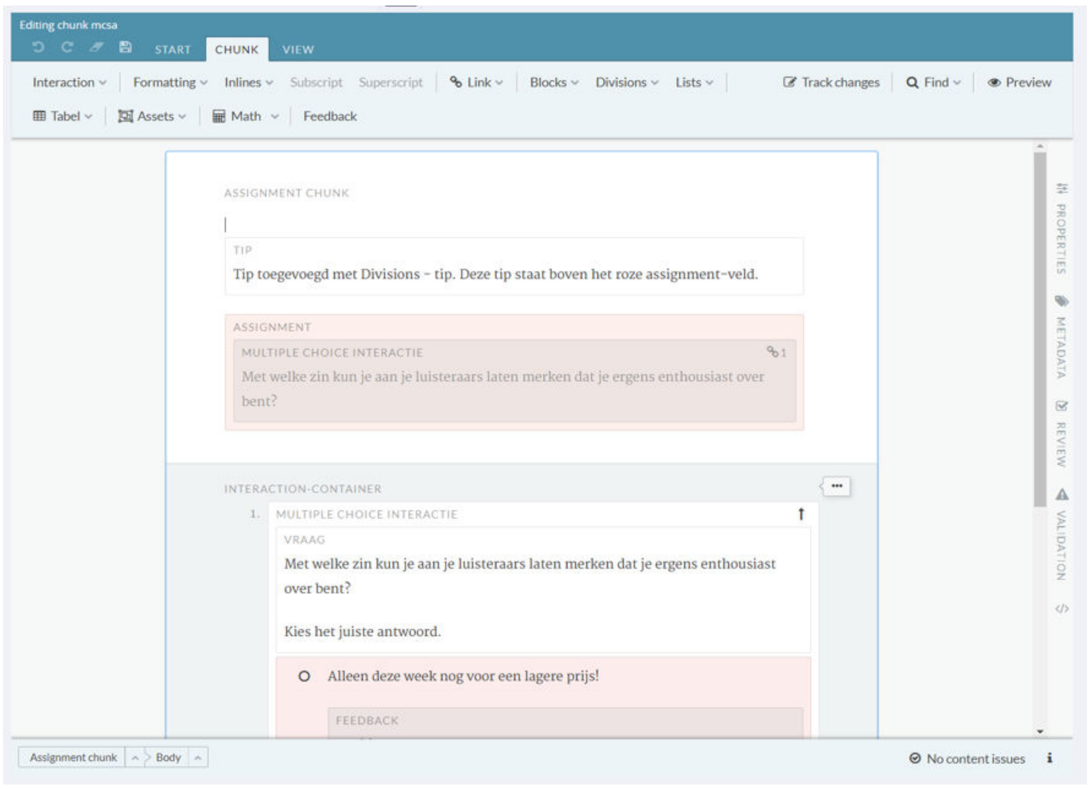
```
- Chunk 1ea32587-26e7-4998-93a2-bae18a8d0657:
 chunk [class="digital-non-assignment-container-chunk" code="201702011600_NN_1_PLPTST01-c00116" id="1ea32587-26e7-
 in feature"]

  chunk-body [id="15e21e1a-b79b-4e6c-e8f2-c0b53bf3891d"]
   p [id="2a67a05f-7525-4309-c5bc-a2109d62e511"]
    Test met nesten features van dezelfde soort.
   div [class="feature1" id="2154e478-2220-4e1b-9cf3-e7ee35159f08"]
    p [id="4e6ed706-e978-49d0-d52e-c071624a4d00"]
     Dit is feature 1.
    div [class="feature1" id="6733fe93-40b1-47d7-e15f-c0e31ba17f33"]
     [undefined-element] (warning): Element div with class "feature1" not allowed here
     p [id="2509d2e3-a204-4db5-8378-7561d84285a0"]
      Met daarin een tweede feature 1. Aanpassing in context zorgt ervoor dat dit een division wordt.
```

  - Content is validated first against the base layer. This involves schema and Schematron checks.

  - When this checks out ok, the content is run through the second layer XSLT stylesheet (about 2000 lines). This tells you whether the content is valid according to the DSVL and, if requested, generates a report with the specifics of the validation errors.

    Such a stylesheet turned out to be an important thing to have, especially in the beginning when your authoring tools are still being developed and you hand-craft most of the example content. And also later, to check the results of the authoring tools.

- We added editors that handled the base layer content-model but could adapt themselves to the DSVL. For structure we created a custom editor. For contents we choose Fonto XML and with the help of Fonto we made it DSVL aware. It now only shows what is allowed according to the DSVL definitions. Toolbars, context menus and metadata forms adapt themselves automagical.



Tip toegevoegd met Divisions - tip. Deze tip staat boven het roze assignment-veld.

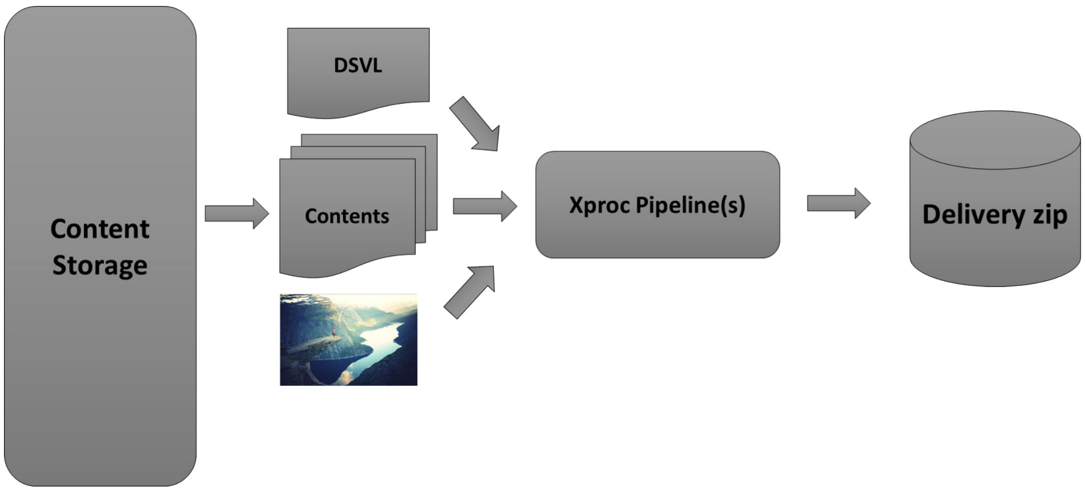
Met welke zin kun je aan je luisteraars laten merken dat je ergens enthousiast over bent?

1. Met welke zin kun je aan je luisteraars laten merken dat je ergens enthousiast over bent?

Kies het juiste antwoord.

Alleen deze week nog voor een lagere prijs!

- We created publication tool-chains (based on XProc) that adapt themselves according to information in the second layer definitions:



One thing I’ve mentioned in the introduction but not touched upon yet is re-use. Yes, having different second layer content models for different products does create silos. However, we have gained quite a lot by basing everything on the same first layer content model. The walls between the silos became much thinner and transforming from one usage to the other is, thanks to the shared base model, much easier.

6. Lesson and takeaways

Now what can we take away from all this and what are important lessons learned:

- The idea worked! We could cater for the differences in content between product families and still use the same underlying basic model and the same tools and tool-chain.

- The language you use for your second-layer modelling must be a true DSVL. It is important that it "talks" about the contents in terms key users (in our case publishers and editors) understand.

  Now of course we didn't expect the key users to be able to make up all this XML themselves, we had to train some specialists to do exactly that. But being able to talk about this in familiar terms was very useful.

- Don't over-engineer the DSVL. It is very tempting to go for being able to define every possible structure and exception. And before you know it you have re-invented XML Schema or RelaxNG…

  Try to find a balanced middle ground between defining everything and language simplicity.

  As a side-note: We had some complicated rules that would have been extremely hard to model in the DSVL but were simple to explain in written instructions based on what the DSVL could already provide as guidance.

- On the other hand, don't under-engineer the DSVL. For instance, in our design it turned out that we had to repeat the same blocks of definitions over and over again which is never a good idea. A little bit of abstraction and type definitions might be necessary.

- Taking the previous two together we get to the obvious open door: Allow for refactoring of the DSVL design (or better: any design) in your project.

- You absolutely need a way to check whether your content conforms to the DSVL, so you'll have to write software for that.

7. Conclusion

Multi-layered content modelling can be a useful solution when you have lots of product families to support with different content models. It allows you to customize things more easily and will decrease the necessary cost for the tool chain.

Combining graph and tree: writing SHAX, obtaining SHACL, XSD and more

Hans-Juergen Rennau
parsQube GmbH
<hrennau@yahoo.de>

Abstract

The Shapes Constraint Language (SHACL) is a data modeling language for describing and validating RDF data. This paper introduces SHAX, which is an XML syntax for SHACL. SHAX documents are easy to write and understand. They cannot only be translated into executable SHACL, but also into XSD describing XML data equivalent to the RDF data constrained by the SHACL model. Similarly, SHAX can be translated into JSON Schema describing a JSON representation of the data. SHAX may thus be viewed as an abstract data modeling language, which does not prescribe a concrete representation language (RDF, XML, JSON, …), but can be translated into concrete models validating concrete model instances.

Keywords: RDF, XML, SHACL, SHAX, XSD, JSON Schema, data modeling, data validation

1. Introduction

RDF ([7], [8]) defines a unified view of information which enables the integration of wildly heterogenous data sources into a uniform graph of RDF nodes. A similar quality has XML. If understood as a data model and technology, rather than a syntax, it can translate tree-structured documents irrespective of the domain of information, the vocabulary used and even the media type (XML, HTML, JSON, CSV, …) into a uniform forest of XDM nodes. Thus both, RDF and XML, offer us the translation of heterogeneous information landscapes into homogeneous spaces of nodes, amenable to powerful tools for addressing, navigating and transforming the contents of these spaces. In spite of the similarities, though, both technologies have different and complementary key strengths, and an integration of RDF and XML technologies is a very promising goal.

A key aspect of integration is how we think about and model RDF data and XML data. Integration might be built upon a careful alignment of these concepts and models, but we must acknowledge a serious mismatch between the major data modeling languages used to describe RDF (RDFS [9] and OWL [5], [6]) and XML data (XSD [16]), respectively. RDF languages are designed for inferencing, and XML modeling concentrates on data validation. The differences do not only pose technical problems, they also reflect quite different mindsets prevailing in the RDF and XML communities.

A new situation has been created by the recent standardization of the Shapes Constraint Language (SHACL, [10]), which is an RDF data modeling and validation language. This paper attempts to identify new possibilities and takes a step towards using them. It introduces SHAX (SHAcl+Xml), which is essentially an XML syntax for SHACL, but might also be regarded as a new and very simple data modeling language which can be translated into both, SHACL and XSD, and thus target RDF and XML data alike. SHAX models are simple to write and to understand, and they may narrow the gap between RDF and XML both conceptually and technically. In particular, they can provide a platform on which to build model-driven translations between RDF graphs and XML trees.

2. RDF and XML: one overarching abstraction

Integration of RDF and XML can be inspired by a clear perception of their essential relationship. At first sight, RDF seems to take a totally unique and “lonely” view of information, reducing it to the atomic unit of a triple: subject, predicate, object. However, when grouping these primary units by subject resource, a secondary unit emerges, which is an entity and its properties. Similarly, a complex XML element can be viewed as an entity enclosing information content. Both, properties and content amount to an itemized description of the entity. Through the equating of properties and content items, one approaches an overarching abstraction: an *information object*. A brief, informal account follows:

- Object abstraction
  - An object is a set of properties
  - A property has a name and one or more values
  - A property value is either a data value or an object value
- XML view
  - Object = complex-typed element
  - Property = attribute or child element
  - Property name = attribute or child element name
  - Property data value = attribute or a simple-typed child element
  - Property object value = complex-typed child element
- RDF view
  - Object = IRI or blank node
  - Property = predicate
  - Property name = predicate IRI
  - Property data value = a literal
  - Property object value = IRI or blank node

Compared to XML, RDF is marked by key strengths. It captures information in an *impartial* way: RDF predicates do not presuppose that the subject is "larger", more general or more interesting than the object. XML represents a partial view, as the containment relationship is arbitrarily directed (does a book contain authors or an author books?), usually reflecting an application-based perspective.

A second aspect is the *amphibian* nature of RDF: while it allows to perceive a solid structure composed of objects (see above), RDF data is at the same time a mere collection of triples, without any primary structure whatsoever. It is like a liquid of information, and this implies a huge benefit: RDF graphs can be merged. Finally, RDF is a natural representation of *complex graphs* (where anything can be related to anything), and such graphs are a natural representation of most non-trivial systems.

XML, however, has unrivalled power of its own. Tree structure is a natural response of the human mind to complexity, as it enables the perception of a coherent whole. It allows a switching between alternative levels of detail. Due to the "spatial" organisation of tree content, navigation can be expressed with superb elegance (XPath). Leveraging this navigation engine, the transformation of trees into all kinds of artifacts – including graphs – is accomplished with amazing ease (XQuery, XSLT).

RDF and XML functioning in cooperation may be regarded as a technological dream team. RDF excels as the organisation of "potentially useful information"; XML excels as the organisation of "ready-for-use" information. How to combine these fundamental strengths? Ultimately, both technologies are about shaping and processing datasets. If datasets can be viewed as instances of data models, the integration of RDF and XML might leverage the knowledge pent up in these models. What is clearly called for is model-driven approaches.

3. RDF and XML: the challenge of integration

The integration of RDF and XML – or, more generally, graph and tree technology - has many interesting aspects. For example, if RDF data describe XML resources in terms of various metadata, XML navigation of a document forest (e.g. the internet) might be supported by SPARQL queries mapping conditions on resource metadata to resource URIs.

Distributed software components communicate via messages, which are usually tree-structured datasets. If the information required by a software component is RDF-based, the typical task at hand is thus the transformation of RDF graph data into an XML or JSON tree. Inversely, if an update of RDF data is necessary and the data is supplied by messages, the transformation of trees into graph data is required.

Remembering the appropriateness of RDF to serve as an unbiased representation of "potentially useful information", the translation of RDF data into a ready-for-use form inspires great interest. For the purpose of this paper, we concentrate on the transformation of trees into graphs and of graphs into trees. Such transformation can always be accomplished by custom code. But given the close relationship between the RDF and XML models (argued in the previous section), transformation should be facilitated by standards or products. Table 1 compiles some approaches to the transformation between RDF and XML data.

Table 1. Standards and products supporting the transformation between RDF and XML data. T2G = „tree to graph", G2T = „graph to tree".

Name	Kind	Direction	Remark
RDFa	W3C recommendation	T2G	Defines markup used for embedding RDF triples in HTML and XML content
GRDDL	W3C recommendation	T2G	Defines the identification of a transformation resource (e.g. XSLT) to be used for extracting RDF triples from HTML or XML content
JSON-LD	W3C recommendation	T2G, G2T	Defines the flexible representation of RDF data by JSON trees, thus transformations in both directions
NIEM	Industry standard	T2G	Defines the equivalence of NIEM XML data with a set of RDF triples
SWP	Commercial product	G2T	A framework for creating HTML and XML content containing and controlled by RDF data
GraphQL	OpenSource product	G2T	Not concerned with RDF and XML, but with abstract representations of graphs and trees: defines the flexible transformation of graphs into trees

It is interesting to note that - apart from GraphQL ([1]) - none of these approaches builds on a data model describing the graph data in terms of type definitions that might be aligned with the building blocks of trees. Equally interesting is the fact that GraphQL is not concerned with RDF and XML data, but with a data source described by an abstract type system and a data target described by an abstract tree model. These observations have an explanation: before SHACL, there was no standardized data modeling language fit for supporting the transformation from/ into graph data.

The advent of SHACL creates new possibilites. For the first time, integration might leverage an RDF data modeling language which can be aligned with an

XML data modeling language - perhaps enabling the alignment of RDF data model components with XML data model components.

4. A short introduction to SHACL

The W3C recommendation ([10]) introduces SHACL as a “language for describing and validating RDF graphs.” It is defined as an RDF vocabulary which can be used to describe data structures, data types and business rules. The key abstractions are constraints and shapes. A *constraint* is a condition which particular RDF nodes must meet. A *shape* is a set of constraints, which jointly must be adhered to. A shape can be thought of as a type, which may be a simple data type or an object type for resources with properties.

Let us consider a very simple example and try to model RDF data describing flight bookings. We decide that a flight booking is represented by a resource belonging to the RDF class `e:FlightBooking`. We want to introduce several constraints: a flight booking has …

- exactly one `e:BookingDate` property, which is a date
- exactly one `e:BookingID` property, which is a string with length >= 12 and <= 40
- an optional `e:BookingChannel`, which is an integer
- one or more `e:OperatingAirlines`, resources (IRIs or blank nodes) which have …
  - exactly one `e:AirlineCode` property, which is a string of two uppercase letters
  - exactly one `e:AirlineName` property, which is a string

This model is a fit for the following RDF graph, represented in Turtle syntax:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e:   <http://example.org/ns/model#> .
@prefix o:   <http://example.org/ns/objects#> .

o:FB101  a                  e:FlightBooking ;
         e:BookingChannel   2 ;
         e:BookingDate      "2017-12-31"^^xsd:date ;
         e:BookingID        "123456789XYZ" ;
         e:OperatingAirline o:AL902 , o:AL901 .

o:AL902  e:AirlineCode      "KL" ;
         e:AirlineName      "KLM - Royal Dutch Airlines" .

o:AL901  e:AirlineCode      "LH" ;
         e:AirlineName      "Lufthansa" .
```

The graph can be validated with a SHACL model, which is itself an RDF graph. Here comes a SHACL model encoding the description given above, represented in Turtle syntax.

```
@prefix sh:   <http://www.w3.org/ns/shacl#> .
@prefix xsd:  <http://www.w3.org/2001/XMLSchema#> .
@prefix e:    <http://example.org/ns/model#> .

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [
      sh:path e:BookingID ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:BookingIDType ;
   ] ;
   sh:property [
      sh:path e:BookingDate ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xsd:date ;
   ] ;
   sh:property [
      sh:path e:BookingChannel ;
      sh:maxCount 1 ;
      sh:datatype xsd:integer ;
   ] ;
   sh:property [
      sh:path e:OperatingAirline ;
      sh:minCount 1 ;
      sh:node e:AirlineType ;
   ] .

e:AirlineType
   a sh:NodeShape ;
   sh:property [
      sh:path e:AirlineCode ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:AirlineCodeType ;
   ] ;
   sh:property [
      sh:path e:AirlineName ;
      sh:maxCount 1 ;
```

```
        sh:datatype xsd:string ;
     ] .

  # data types
  # ==========
  e:BookingIDType
     a sh:NodeShape ;
     sh:datatype xsd:string ;
     sh:minLength 12 ;
     sh:maxLength 40 .

  e:AirlineCodeType
     a sh:NodeShape ;
     sh:datatype xsd:string ;
     sh:pattern "^[A-Z]{2}$" .
```

This structure can be read as follows: `e:FlightBookingType` is a shape modeling the members of a particular RDF class (`e:FlightBooking`), and each `sh:property` [...] statement describes a mandatory or optional property of those resources, with a property name identified by `sh:path`:

```
  e:FlightBookingType
     a sh:NodeShape ;
     sh:targetClass e:FlightBooking ;
     sh:property [sh:path: e:BookingID ; …];
     sh:property [sh:path: e:BookingDate ; …];
     sh:property [sh:path: e:BookingChannel ; …];
     sh:property [sh:path: e:OperatingAirline ; …];>
```

Further settings within the `sh:property[…]` statements constrain the property in terms of cardinality and value type, e.g.:

```
  sh:property [
        sh:path e:BookingDate ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:date ;
     ] ;>
```

constrains the `e:BookingDate` property to have exactly one value which is a date.

The example demonstrated capabilities which are similar to those of the XSD language. XSD describes data structure in terms of *element contents* (attributes and child elements), whereas SHACL describes data structure in terms of *resource properties* (data and object properties). XSD constrains content items in terms of name, cardinality and a type which is either a simple data type or a model of (nested) element contents. SHACL constrains properties in terms of name, cardinality and a type which is either a simple data type or a model of (nested) resource properties.

Accepting the analogy between element contents and resource properties, one realizes that most capabilities of XSD can be expressed in SHACL. But SHACL has still more to offer. First, it supports constraints targeting *pairs* of properties – e.g. stating that the value of one property must be less than the value of another property. SHACL also allows the construction of complex constraints which are a *logical combination* of other constraints (`sh:and`, `sh:or`, `sh:not`, `sh:xone`). A property can for example be described as either a string or one of several alternative object types.

SHACL models can describe “virtual” properties, which do not correspond to a simple property IRI, but are defined as a property path expression. The following are a few examples:

```
sh:path ( ex:address ex:street )
sh:path [ sh:inversePath ex:parent ]
sh:path [ sh:alternativePath ( ex:father ex:mother ) ]
sh:path ( ex:reads [ sh:zeroOrMorePath ex:cites ] )
```

Perhaps most importantly, SHACL supports the use of *SPARQL queries* for expressing constraints of virtually unlimited complexity and specificity. Such expressions could be used to capture XSD features not explicitly supported (like choice groups, identity constraints and XSD 1.1 assertions), as well as arbitrary business rules which are typically validated by Schematron models.

To summarize, SHACL combines the capabilities of XSD and Schematron. Besides, it allows the construction of complex constraints and supports virtual properties.

5. SHAX – motivation

The SHACL language enables a modeling of RDF data which makes them understandable for anyone familiar with data modeling, not requiring a deep understanding of RDF. Such models facilitate the development of applications processing or creating RDF data. Apart from this mental level, SHACL-based validation enables elaborate guarantees of data quality, which may reduce code complexity.

How to use these potential benefits in practice? Several problems must be overcome:

- SHACL is defined as an *RDF vocabulary*, but only few IT professionals are familiar with RDF
- *Tool support* for processing RDF data is rather limited – transformation of SHACL data into other artifacts tends to be difficult

- A *grammar-oriented data model* - describing structured content in terms of a simple and generic content model – is not always straightforward to express because:
  - Choice groups require convoluted constructs (combining `sh:xone`, `sh:not` and `sh:and`)
  - The description of a single property may be spread over several `sh:property` statements

These difficulties might be overcome by an *XML representation* of SHACL models. It offers a simple tree structure which is trivial to write and read. An XML representation can be translated into object oriented structures (e.g. via JAXB [2]) or directly processed by powerful processing and transformation languages (XQuery [15] and XSLT [17]). The convolutions sometimes required to express basic grammatical constructs (choice groups and properties-as-building-blocks) can be hidden behind the façade of a straightforward expression of intent.

The promise of an XML representation goes beyond this simplification of writing, reading and processing of SHACL models. It may pave the way for *abstract data models*, which define data structures and data types, but refrain from prescribing the representation language, in particular RDF versus non-RDF expression like XML or JSON. This possibility could be realized by transforming the XML representation of a SHACL model alternatively into a SHACL model (applicable to RDF data) or an instance of a non-RDF data modeling language, like XSD ([16]) or JSON Schema ([3]), which describes a non-RDF representation of RDF content.

If the XML representation is designed to increase the abstraction level of a SHACL model, a single XML source might be used to generate alternative SHACL styles. An example is the use of local versus global shapes, comparable to XSD styles preferring local versus global types. Last but not least, an XML representation might prove a convenient platform for augmenting model components with *metadata* – for example metadata specifying a data source for constructing the item, or a data destination for dispatching the item.

The remainder of this paper introduces **SHAX** (= SHAcl+Xml), which can be regarded as an XML representation of (some core parts of) the SHACL language, or, alternatively, as a new data modeling language which may be translated into SHACL, XSD and JSON Schema.

6. SHAX – introductory examples

A SHAX model describes *object types* in terms of their *properties*. The following SHAX model describes a single object type (`e:FlightBookingType`) which represents a flight booking:

```
<shax:model defaultCard="1"
    xmlns:shax="http://shax.org" xmlns:e="http://example.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- object types
         ============ -->
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
    </shax:objectType>

    <!-- data types
         ========== -->
    <shax:dataType name="e:BookingIDType"
                   base="xsd:string" minLen="12" maxLen="40"/>
</shax:model>
```

Flight bookings have three properties:

- A `BookingID`, which is a string with a length between 12 and 40 characters
- A `BookingDate`, which is an instance of `xsd:date`
- A `BookingChannel`, which is an instance of `xsd:integer`

The `BookingChannel` is optional (@card="?"), and the other properties are mandatory (@defaultCard="1"). The model is matched by the following RDF data:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e:   <http://example.org/ns/model#> .
@prefix o:   <http://example.org/ns/objects#> .

o:FB101 a                 e:FlightBooking ;
        e:BookingChannel  2 ;
        e:BookingDate     "2017-12-31"^^xsd:date ;
        e:BookingID       "123456789XYZ" .>
```

After downloading the SHAX processor from github ([14]), we can translate our SHAX model into a SHACL model. The command line call:

```
    shax "shacl?shax=flightBooking01.shax"
```

produces the following model:

```
@prefix sh:  <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e:   <http://example.org/ns/model#> .

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
```

```
      sh:targetClass e:FlightBooking ;
      sh:property [
         sh:path e:BookingID ;
         sh:minCount 1 ;
         sh:maxCount 1 ;
         sh:node e:BookingIDType ;
      ] ;
      sh:property [
         sh:path e:BookingDate ;
         sh:minCount 1 ;
         sh:maxCount 1 ;
         sh:datatype xsd:date ;
      ] ;
      sh:property [
         sh:path e:BookingChannel ;
         sh:maxCount 1 ;
         sh:datatype xsd:integer ;
      ] .

   # data types
   # ==========
   e:BookingIDType
      a sh:NodeShape ;
      sh:datatype xsd:string ;
      sh:minLength 12 ;
      sh:maxLength 40 .>
```

In order to see this model at work, we launch the "SHACL playground" (http://shacl.org) in a browser and copy model and instance data into the areas labeled "Shapes Graph" and "Data Graph". Using the "Update" buttons beneath both areas, we observe that the data are valid against the model. We also observe how various manipulations of instance data or the model cause validation errors (see area "Validation Report", where any error reports appear).

Having seen that our SHAX model can be compiled into a functional SHACL model, we proceed to extend our model, adding a property OperatingAirline. This is an *object property*, as the property values are objects which have their own properties: a mandatory code (two uppercase letters) and an optional name (string). Here comes the new version:

```
   <shax:model defaultCard="1"
       xmlns:shax="http://shax.org/ns/model"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:e="http://example.org/ns/model">

       <!-- object types
            ============ -->
```

```
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
        <e:OperatingAirline card="+" type="e:AirlineType"/>
    </shax:objectType>

    <shax:objectType name="e:AirlineType">
        <e:AirlineCode type="e:AirlineCodeType"/>
        <e:AirlineName card="?" type="xsd:string"/>
    </shax:objectType>

    <!-- data types
         ========== -->
    <shax:dataType name="e:BookingIDType"
                   base="xsd:string" minLen="12" maxLen="40"/>
    <shax:dataType name="e:AirlineCodeType"
                   base="xsd:string" pattern="^[A-Z]{2}$"/>

</shax:model>
```

Note the cardinality constraint (@card="+"), which means “one or more” values. In the final step we add a `Customer` property:

```
<shax:model defaultCard="1"
    xmlns:shax="http://shax.org/ns/model"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:e="http://example.org/ns/model">

    <!-- properties
         ========== -->
    <shax:property name="e:FlightBooking" type="e:FlightBookingType"/>

    <!-- object types
         ============ -->
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
        <e:OperatingAirline card="+" type="e:AirlineType"/>
        <e:Customer type="e:CustomerType"/>
    </shax:objectType>

    <shax:objectType name="e:AirlineType" class="e:Airline">
        <e:AirlineCode type="e:AirlineCodeType"/>
        <e:AirlineName card="?" type="xsd:string"/>
    </shax:objectType>
    <shax:objectType name="e:CustomerType" class="e:Customer">
        <e:LastName type="xs:string"/>
        <e:FirstName type="xs:string"/>
        <shax:choice>
            <e:PassportNumber type="xsd:string"/>
            <shax:pgroup>
                <e:LoyaltyProgramCode type="e:LoyaltyProgramCodeType"/>
                <e:LoyaltyProgramMemberID type="xsd:integer"/>
            </shax:pgroup>
        </shax:choice>
    </shax:objectType>

    <!-- data types
         ========== -->
    <shax:dataType name="e:BookingIDType"
                   base="xsd:string" minLen="12" maxLen="40"/>
    <shax:dataType name="e:AirlineCodeType"
                   base="xsd:string" pattern="^[A-Z]{2}$"/>
    <shax:dataType name="e:LoyaltyProgramCodeType"
                   base="xsd:integer" min="1" max="999"/>
</shax:model>
```

Here comes a conforming RDF graph, this time represented in JSON-LD:

```
{
   "@context": {
      "xsd": "http://www.w3.org/2001/XMLSchema#",
      "e": "http://example.org/ns/model#",
      "o": "http://example.org/ns/objects#"
   },
   "@graph": [
      {
         "@id": "o:FB101",
         "@type": "e:FlightBooking",
         "e:BookingID": "123456789XYZ",
         "e:BookingDate": {"@value": "2017-12-31", "@type": "xsd:date"},
         "e:BookingChannel": 2,
         "e:OperatingAirline": [
            {
               "@id": "o:AL901",
               "e:AirlineCode": "LH",
               "e:AirlineName": "Lufthansa"
            },
            {
               "@id": "o:AL902",
               "e:AirlineCode": "KL",
               "e:AirlineName": "KLM - Royal Dutch Airlines"
            }
         ],
         "e:Customer": {
            "e:LastName": "Perez",
            "e:FirstName": "Deborah",
            "e:LoyaltyProgramCode": 800,
            "e:LoyaltyProgramMemberID": 81557
         }
      }
   ]
}
```

The SHAX model is translated into a SHACL model which is more than three times as long (see Appendix A). Note that the SHACL model does implement the choice group structure (see the `sh:xone` element), but it does not make this structure explicit.

To summarize, this section demonstrated various features of SHAX models, including object and data types, cardinality constraints and the grammatical structure of a choice group. The next section gives a more systematic overview of the SHAX language.

7. SHAX – building blocks

A SHAX model is represented by a `shax:model` element which contains various kinds of top-level model components:

- `shax:objectType` – defines an object type
- `shax:dataType` – defines a data type
- `shax:property` – defines a property which can be referenced by object types

Models can import other models, using `shax:import` elements.

7.1. Element `shax:model`

A SHAX model is represented by a `shax:model` element. It contains elements importing other models (`shax:import`), as well as elements representing top-level model components. The `shax:model` element has an optional @defaultCard attribute which sets a default value for cardinality constraints (@card attributes, see below). In the absence of an @defaultCard attribute, the default cardinality is “exactly one”. Note that a SHAX model does not have a target namespace.

7.2. Element `shax:objectType`

An object type is a named model of resources. Above all, it is a description of the properties which adhering resources may have. Besides, the model can constrain the resources to belong to one or more (RDF) classes and to have a certain node kind (IRI or blank node). For example:

```
<shax:objectType name="e:BookingType" class="e:Booking" nodeKind="IRI">…
```

The possible properties of an object are described by *property declarations*, which are represented by elements named after the property in question. The @card attribute expresses a cardinality constraint (?, *, +, i, i-j), and the @type attribute constrains the property values to have a particular object or data type. A data type may be a built-in type (if from the XSD namespace) or a model-defined data type (see below, `shax:dataType`). For example:

```
<shax:objectType name="e:BookingType" class="e:Booking" nodeKind="IRI">
    <e:BookingID type="e:BookingIDType"/>
    <e:BookingDate type="xsd:date"/>
    <e:BookingChannel card="?" type="xsd:integer"/>
</shax:objectType>
```

Note that the order of property declarations is significant information. An object type can *extend* another one, which means that the property declarations of the extended type can be thought of as preceding the properties of the extending type. For example:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:OperatingAirline card="+" type="e:AirlineType"/>
</shax:objectType>
```

A property declaration may also be a reference to a globally defined property. Reference is implicit, recognized by the absence of a @type attribute. The object or data type of the property is specified by the referenced property declaration:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:OperatingAirline card="+"/>
</shax:objectType>
    …
<shax:property name="e:OperatingAirline" type="e:AirlineType"/>
```

Note that property references can be used for defining extension points, described below. An object model may include *choices* of properties, represented by a `shax:choice` element with one child element per alternative. An alternative is either a single property or a group of properties (wrapped in a `shax:pgroup` element). Choices can be nested. For example:

```
<shax:choice>
     <e:PassportNumber type="xsd:string"/>
     <shax:pgroup>
         <e:LoyaltyProgramCode type="e:LoyaltyProgramCodeType"/>
         <shax:choice>
             <e:LoyaltyProgramMemberID type="xsd:integer"/>
```

```
            <e:PreliminaryMemberID type="xsd:string"/>
        <shax:choice>
    </shax:pgroup>
</shax:choice>
```

7.3. Element `shax:property`

A `shax:property` element defines a property in a *global* way, independent of the object type which uses it. The declaration specifies a property name (@name) and a value type (@type), which is the IRI of an object or data type. An optional @substitutes attribute specifies the names of one or more (global) properties which the given property may *substitute*. This means that references to the specified properties are also matched by properties adhering to the substituting property declaration. Note the constraint that the substituting property declaration must have a type which extends or restricts the types of all substitutable property declarations.

Substitution can be used to define *extension points*, a property representing a logical choice between the property itself and all other property declarations that are able to substitute it. For example:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:Airline card="+"/>
</shax:objectType>
   …
<shax:property name="e:Airline" type="e:AirlineType"/>
<shax:property name="e:OperatingAirline" type="e:AirlineType"
               substitutes="e:Airline"/>
<shax:property name="e:MarketingAirline" type="e:MarketingAirlineType"
               substitutes="e:Airline"/>
   …
<shax:objectType name="e:MarketingAirlineType" extends="e:AirlineType">…
```

7.4. Element `shax:dataType`

A data type is a set of constraints applicable to data values. The concept of a data type follows the approach taken by XSD:

- A data type is either atomic, or a list type, or a union type
- An atomic type associates a base type with constraints, called facets
- A list type describes an ordered sequence of atomic items (in RDF represented as `rdf:List`)
- A union type is defined as a choice between two or more data types

A data type is represented by a `shax:dataType` element. An atomic type has a @base attribute specifying the base type, and most facets are specified by attributes (@pattern, @len, @minLen, @maxLen, @min, @minEx, @max, @maxEx). The enumeration facet, which specifies a list of all possible values, is represented by `shax:value` child elements. Examples:

```
<shax:dataType name="e:AirlineCodeType"
               base="xsd:string" pattern="^[A-Z]{2}$"/>

<shax:dataType name="e:GenderType" base="xsd:string">
    <shax:value>male</shax:value>
    <shax:value>female</shax:value>
</shax:dataType/>
```

A list type definition specifies the item type with an @itemType attribute; optional @minSize, @maxSize and @size attributes constrain the number of list items. A union type definition specifies the alternative types with a @memberTypes attribute. Examples:

```
<shax:dataType name="e:AirlineCodesType"
               itemType="xsd:AirlineCodeType"
               minSize="1">

<shax:dataType name="e:ExtendedCurrencyType"
               memberTypes="xsd:CurrencyCodeType xsd:int">
```

8. Translation into concrete data modeling languages

SHAX can be used for constructing *abstract data models* – models defining structures of information content, but not prescribing the data format of concrete representations. In order to allow *validation* of data against a SHAX model, data can be represented in different formats:

- RDF (any serialization)
- XML
- JSON

For each of these formats, a SHAX model can be translated into an appropriate data validation language (SHACL, XSD, JSON Schema). Note that these translations may involve a certain loss of information. The sequence of properties, for example, is not preserved when translating into JSON Schema, and when translating into SHACL, although it is preserved as information (using `sh:order`), it loses its impact on validation. Another example is property pair constraints (e.g. @shax:lessThan) which are only preserved when translating into SHACL.

8.1. SHAX into SHACL

The translation into a SHACL model can be performed in two different styles. These styles are similar to the XSD styles using only global types (“flat” style) or only local types (“deep” style):

- Flat – every SHAX type reference (@type) is translated into a `sh:node` constraint
- Deep – every SHAX type reference is replaced by the constraints defined by the referenced type

The flat style ensures a well-modularized SHACL model. A disadvantage concerns the error messages in case of validation errors, which are less specific. Messages from flat/deep SHACL:

```
sh:resultMessage "Value does not have shape e:BookingIDType" ;
sh:resultMessage "Value has less than 12 characters" ;
```

8.2. SHAX into XSD

The XSD(s) obtained from a SHAX model describe XML documents which are a canonical representation of a model instance. The definition of this representation (not formally specified here) has been inspired by the definition of XML/RDF equivalence described in the NIEM Naming and Design Rules ([4], section 5.6). Key points are the mapping of property IRIs to element names and the representation of resource IRIs by dedicated attributes (@shax:IRI). The following listing shows an XML representation of the RDF data shown above (Model instance, RDF).

```
<e:FlightBooking xmlns:shax="http://shax.org/ns/model"
                 xmlns:e="http://example.org/ns/model"
                 xmlns:o="http://example.org/ns/objects"
                 shax:IRI="o:FB101">
    <e:BookingID>123456789XYZ</e:BookingID>
    <e:BookingDate>2017-12-31</e:BookingDate>
    <e:OperatingAirline shax:IRI="o:AL901">
        <e:AirlineCode>LH</e:AirlineCode>
        <e:AirlineName>Lufthansa</e:AirlineName>
    </e:OperatingAirline>
    <e:OperatingAirline shax:IRI="o:AL902">
        <e:AirlineCode>KL</e:AirlineCode>
        <e:AirlineName>KLM - Royal Dutch Airlines</e:AirlineName>
    </e:OperatingAirline>
    <e:Customer>
        <e:LastName>Perez</e:LastName>
        <e:FirstName>Deborah</e:FirstName>
        <e:LoyaltyProgramCode>800</e:LoyaltyProgramCode>
        <e:LoyaltyProgramMemberID>81557</e:LoyaltyProgramMemberID>
    </e:Customer>
</e:FlightBooking>
```

The translation of SHAX into XSD is straightforward:

- `shax:dataType` is translated into `xs:simpleType`
- `shax:objectType` is translated into `xs:complexType`
- `shax:property` is translated into `xs:element`, which is top-level
- Property elements are translated into `xs:element` defined within complex type definitions; the element declaration has an @type or @ref attribute, dependent on whether or not the property element has a @type attribute

See appendix (Appendix B) for a listing of the XSD obtained for the SHAX model described above (SHAX model example).

8.3. SHAX into JSON Schema

The JSON Schema obtained describes JSON documents which are a straightforward representation of a model instance. A somewhat primitive style is assumed which ignores namespaces and uses only local names. Investigation of an advanced representation model using JSON-LD is a work in progress. See appendix (Appendix C) for the JSON Schema obtained from the SHAX model described above (SHAX model example).

9. SHAX - implementation

A SHAX processor accomplishes the translation of SHAX models into SHACL, XSD and JSON Schema models. An implementation of a SHAX processor is available on github (`https://github.com/hrennau/shax`). The processor is provided as a command line tool. Example calls:

```
shax shacl?shax=airportSystem.shax
shax xsd?shax=airportSystem.shax
shax jschema?shax=airportSystem.shax
```

10. Prospect: SHAX for RDF-XML integration

Any transformation of RDF data into XML documents can be accomplished by custom code. However, in situations where source and/or target data are described by data models, it would be unsatisfactory if custom code were the only option. A model-driven alternative might define a way how available descriptions of source or target structures can control the transformation and thus significantly reduce the effort. A detailed investigation of such possibilities is beyond the scope of this paper, but an example is sketched. The general goal is to enable transformation implemented in a declarative way, that is, modeling the transformation, rather than coding it.

A transformation of RDF data into an XML representation might be defined in terms of an EFRUG model, which expresses transformation as the successive application of operations to RDF source data. The operations have a generic definition and respond to model facts in a configurable way. Starting with a root resource to be transformed …

- **EF** – expand and filter the resource properties recursively, ignoring irrelevant properties and expanding object property values by a nested representation of their own property values
- **R** – rename the properties, starting with a canonical translation into a qualified name and replacing those qualified names which are not satisfactory
- **U** – unwrap nodes, removing inner nodes and connecting their parent nodes to their child nodes, where simplification is desirable
- **G** – group nodes by introducing intermediate nodes which wrap subsets of related siblings, where additional structure is desirable

Note that the key step is a filtered iteration over properties, combined with recursive expansion of object properties. It is easy to implement such an iteration, given a graph model expressed in SHAX, and it should not be difficult to render the filtering configurable, perhaps using whitelists and blacklists of regular expressions.

11. Discussion

The SHAX language as described in this paper is only a preliminary version serving as a proof of concept. It focuses on a subset of the SHACL language which enables the definition of a traditional object type system, as this subset is key for aligning RDF models and tree models like XSD. Future work should strive for a steady increase of the supported capabilities of SHACL. It is to be hoped that the simplicity and clarity of the SHAX language will be preserved in the process.

We regard the SHACL language as an important advance, which makes the introduction of an abstract data modeling language possible – a language capable of describing graph and tree encoded information. Key aspects of this language are the definition of equivalence between RDF and non-RDF representations, and the transformation of the abstract model into concrete, validating models for each major representation language.

The greatest problem ahead may not be a technical one, but a cultural one: to arouse in both communities, RDF and XML, an awareness of how complementary their technologies are, of how incomplete they are without each other.

A. SHACL model obtained from SHAX

The following listing shows the result of translating a SHAX model into SHACL.

Tip: You can experiment with this model here: `http://shacl.org`

- SHAX source model - see: SHAX model example

- Model instance, RDF - see: Model instance, RDF

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix xs:  <http://www.w3.org/2001/XMLSchema#> .
@prefix e: <http://example.org/ns/model#> .

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [
      sh:path e:BookingID ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:BookingIDType ;
   ] ;
   sh:property [
      sh:path e:BookingDate ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xsd:date ;
   ] ;
   sh:property [
      sh:path e:BookingChannel ;
      sh:maxCount 1 ;
      sh:datatype xsd:integer ;
   ] ;
   sh:property [
      sh:path e:OperatingAirline ;
      sh:minCount 1 ;
      sh:node e:AirlineType ;
   ] ;
   sh:property [
      sh:path e:Customer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:CustomerType ;
   ] .

e:AirlineType
   a sh:NodeShape ;
   sh:targetClass e:Airline ;
   sh:property [
      sh:path e:AirlineCode ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:AirlineCodeType ;
   ] ;
   sh:property [
      sh:path e:AirlineName ;
      sh:maxCount 1 ;
      sh:datatype xsd:string ;
   ] .

e:CustomerType
   a sh:NodeShape ;
   sh:targetClass e:Customer ;
   sh:property [
      sh:path e:LastName ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xs:string ;
   ] ;
   sh:property [
      sh:path e:FirstName ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xs:string ;
   ] ;
   sh:xone (
      [
         a sh:NodeShape ;
         sh:not
            [
               a sh:NodeShape ;
               sh:or (
                  [
                     sh:path e:LoyaltyProgramCode ;
                     sh:minCount 1 ;
                  ]
                  [
                     sh:path e:LoyaltyProgramMemberID ;
                     sh:minCount 1 ;
                  ]
               ) ;
            ] ;
         sh:property [
            sh:path e:PassportNumber ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:datatype xsd:string ;
         ] ;
      ]
      [
         a sh:NodeShape ;
         sh:not
            [
               sh:path e:PassportNumber ;
               sh:minCount 1 ;
            ] ;
         sh:property [
            sh:path e:LoyaltyProgramCode ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:node e:LoyaltyProgramCodeType ;
         ] ;
         sh:property [
            sh:path e:LoyaltyProgramMemberID ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:datatype xsd:integer ;
         ] ;
      ]
   ) .

# data types
# ==========
e:BookingIDType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:minLength 12 ;
   sh:maxLength 40 .

e:AirlineCodeType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:pattern "^[A-Z]{2}$" .

e:LoyaltyProgramCodeType
   a sh:NodeShape ;
   sh:datatype xsd:integer ;
   sh:minInclusive 1 ;
   sh:maxInclusive 999 .
```

B. XSD model obtained from SHAX

The following listing shows the result of translating a SHAX model into XSD.

- SHAX source model - see: SHAX model example
- Model instance, RDF - see: Model instance, RDF
- Model instance, XML - see: Model instance, XML

```
<xs:schema xmlns:shax="http://shax.org/ns/model"
           xmlns:e="http://example.org/ns/model"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://example.org/ns/model"
           elementFormDefault="qualified">
  <xs:import namespace="http://shax.org/ns/model"
             schemaLocation="shax.xsd"/>
  <xs:element name="FlightBooking" type="e:FlightBookingType"/>
  <xs:complexType name="FlightBookingType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="BookingID" type="e:BookingIDType"/>
          <xs:element name="BookingDate" type="xs:date"/>
          <xs:element name="BookingChannel" minOccurs="0"
                      type="xs:integer"/>
          <xs:element name="OperatingAirline" maxOccurs="unbounded"
                      type="e:AirlineType"/>
          <xs:element name="Customer" type="e:CustomerType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="AirlineType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="AirlineCode" type="e:AirlineCodeType"/>
          <xs:element name="AirlineName" minOccurs="0" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="CustomerType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="LastName" type="xs:string"/>
          <xs:element name="FirstName" type="xs:string"/>
          <xs:choice>
            <xs:element name="PassportNumber" type="xs:string"/>
            <xs:sequence>
              <xs:element name="LoyaltyProgramCode"
```

```
                            type="e:LoyaltyProgramCodeType"/>
                <xs:element name="LoyaltyProgramMemberID"
                            type="xs:integer"/>
              </xs:sequence>
            </xs:choice>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
    <xs:simpleType name="BookingIDType">
      <xs:restriction base="xs:string">
        <xs:minLength value="12"/>
        <xs:maxLength value="40"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="AirlineCodeType">
      <xs:restriction base="xs:string">
        <xs:pattern value="[A-Z]{2}"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="LoyaltyProgramCodeType">
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="1"/>
        <xs:maxInclusive value="999"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

C. JSON Schema model obtained from SHAX

The following listing shows the result of translating a SHAX model into JSON schema.

Tip: You can experiment with this schema here: https://www.jsonschemavalidator.net/

- SHAX source model - see: SHAX model example
- Model instance, RDF - see: Model instance, RDF

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "type":"object",
  "properties":{
    "FlightBooking":{
      "type":"object",
      "properties":{
        "IRI":{
```

```
  "$ref":"#/definitions/xs:anyURI"
},
"BookingID":{
  "$ref":"#/definitions/a:BookingIDType"
},
"BookingDate":{
  "$ref":"#/definitions/xs:date"
},
"BookingChannel":{
  "$ref":"#/definitions/xs:integer"
},
"OperatingAirline":{
  "type":"array",
  "minItems":1,
  "maxItems":999,
  "items":{
    "type":"object",
    "properties":{
      "IRI":{
        "$ref":"#/definitions/xs:anyURI"
      },
      "AirlineCode":{
        "$ref":"#/definitions/a:AirlineCodeType"
      },
      "AirlineName":{
        "$ref":"#/definitions/xs:string"
      }
    },
    "additionalProperties":false
  }
},
"Customer":{
  "type":"object",
  "properties":{
    "IRI":{
      "$ref":"#/definitions/xs:anyURI"
    },
    "LastName":{
      "$ref":"#/definitions/xs:string"
    },
    "FirstName":{
      "$ref":"#/definitions/xs:string"
    },
    "PassportNumber":{
      "$ref":"#/definitions/xs:string"
    },
              "LoyaltyProgramCode":{
                "$ref":"#/definitions/a:LoyaltyProgramCodeType"
              },
              "LoyaltyProgramMemberID":{
                "$ref":"#/definitions/xs:integer"
              }
            },
            "additionalProperties":false,
            "required":[
              "LastName",
              "FirstName"
            ],
            "oneOf":[
              {
                "required":[
                  "PassportNumber"
                ]
              },
              {
                "required":[
                  "LoyaltyProgramCode",
                  "LoyaltyProgramMemberID"
                ]
              }
            ]
          }
        },
        "additionalProperties":false,
        "required":[
          "BookingID",
          "BookingDate",
          "OperatingAirline",
          "Customer"
        ]
      }
    },
    "definitions":{
      "a:AirlineCodeType":{
        "type":"string",
        "pattern":"^[A-Z]{2}$"
      },
      "a:BookingIDType":{
        "type":"string",
        "minLength":12,
        "maxLength":40
      },
      "a:LoyaltyProgramCodeType":{
        "type":"integer",
        "minimum":1,
        "maximum":999
      },
      "xs:anyURI":{
        "type":"string"
      },
      "xs:date":{
        "type":"string"
      },
      "xs:integer":{
        "type":"integer"
      },
      "xs:string":{
        "type":"string"
      }
    }
  }
```

Bibliography

[1] *GraphQL*. 2017. Facebook Inc.. `http://graphql.org/`

[2] *Java Architecture for XML Binding (JAXB)*. 2017. Java Community Process. `https://jcp.org/en/jsr/detail?id=222`

[3] *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. 2017. Internet Engineering Task Force. `http://json-schema.org/latest/json-schema-validation.html`

[4] *National Information Exchange Model Naming and Design Rules (Version 4.0)*. 2017. NIEM Program Management Office (PMO). `https://reference.niem.gov/niem/specification/naming-and-design-rules/4.0rc1/niem-ndr-4.0rc1.html`

[5] *OWL 2 Web Ontology Language Primer (Second Edition)*. 2012. World Wide Web Consortium (W3C). `https://www.w3.org/TR/owl2-primer/`

[6] *OWL 2 Web Ontology Language Quick Reference Guide (Second Edition)*. 2012. World Wide Web Consortium (W3C). `https://www.w3.org/TR/owl-quick-reference/`

[7] *RDF 1.1 Concepts and Abstract Syntax*. 2014. World Wide Web Consortium (W3C). `https://www.w3.org/TR/rdf11-concepts/`

[8] *RDF 1.1 Primer*. 2014. World Wide Web Consortium (W3C). `https://www.w3.org/TR/rdf11-primer/`

[9] *RDF Schema 1.1*. 2014. World Wide Web Consortium (W3C). `https://www.w3.org/TR/rdf-schema/`

[10] *Shapes Constraint Language (SHACL)*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/shacl/`

[11] *SHACL Advanced Features*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/shacl-af/`

[12] *SHACL JavaScript Extensions*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/shacl-js/`

[13] *SHACL and OWL Compared*. Holger Knublauch. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/shacl-js/`

[14] *A SHAX processor, transforming SHAX models into SHACL, XSD and JSON Schema.*. Hans-Juergen Rennau. 2018. `https://github.com/hrennau/shax`

[15] *XQuery 3.1: An XML Query Language*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xquery-31/`

[16] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 2012. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xmlschema11-1/`

[17] *XSL Transformations (XSLT) Version 3.0*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xslt/`

SML – A simpler and shorter representation of XML

Jean-François Larvoire
Hewlett Packard Enterprise
<jf.larvoire@hpe.com>

Abstract

When XML is used for encoding structured data, one of the things people most often complain about is that XML is more verbose, and harder to read by humans, than most alternatives. This may even cause some of them to abandon XML altogether.

Many alternatives to XML have actually been designed to specifically address this issue. Some are indeed better, being both simple and more powerful. But I think that creating new standards for this reason is missing the point. XML and JSON now dominate the structured data interchanges, and they're not going to be displaced any time soon, even by better alternatives.

Instead, this paper proposes a Simplified representation of XML (SML for short), that is strictly equivalent to XML. Strictly equivalent in the sense that any XML file can be converted to SML, then back into XML, and be binary equal to the initial file. And these SML data files are smaller, and much easier to read and edit by mere humans.

A Tcl script called sml.tcl is available for easily testing that concept, by converting files back and forth between the XML and SML formats. I've been using it advantageously for several years as part of my job. Every time I have to review an unknown XML file, I convert it to SML and open it in a plain text editor. It's arguably even easier to read than JSON. Then, if changes are needed, I make these changes in the SML text, and convert the result back to XML.

Recently, I verified that the full libxml2 test suite can be successfully converted to SML and back, with no change.

Also I'm working on a libxml2 fork that can parse both XML and SML, and output either one at will. A demonstrator is available on GitHub, including a C XML↔SML conversion program called x2s.exe that's 20 times faster than the Tcl script.

Other qualities:

- SML files are noticeably smaller than XML files. Using this format directly for storage or data transfer protocols saves space and network band-

width. This does not require rewriting any XML data creation/consumption routine, but just to insert XML↔SML conversion routines in the pipeline.

- SML is a nice format for serializing and reviewing small file system trees contents, for example the Linux /proc/fs trees.

Limitations:

- The simplification is considerable for structured data trees, but less so for mixed content cases, like in XHTML, DocBook, etc. Although all such mixed files can also be successfully converted to SML and back, the SML version may actually be more complex than the original XML. This is especially the case for XHTML files with markup peppered randomly all over the text. On the other hand, well formatted DocBook converts rather well.

Note: I'm aware that another data format called SML was proposed in 1999. The proposal here has no relationship at all with the other one from 1999. If this homonymy proves to be a problem, I'm open to any suggestion as to a better name.

Keywords: XML, SML, Markup, Serialization, Serialization formats

1. Introduction

I started thinking about alternative views into XML files many years ago because of a personal itch: I needed to repeatedly tweak a complex XML configuration file for a Linux Heartbeat cluster in the lab. No DTDs available. No specialized XML editors installed on that machine. Editing the file using a plain text editor was painful every time.

Why had it to be so? XML is a text format that was supposed to be designed for easy manual edition by humans. And XML proponents actually list this feature as an advantage of XML. Yet XML tags are so verbose that it is a pain to manually review and edit anything but trivial XML files. The numerous XML editors available are a relief, but do not resolve the fundamental problem of XML verbosity when it comes to simply reading the file. (Actually I think their very existence is proof that XML has a problem!)

In the absence of a solution, I avoided using XML for my own projects as much as I could, and kept looking at alternatives, in the hope that one of them would eventually replace XML as the new data exchange standard.

1.1. Alternatives to XML

1.1.1. Distinct syntaxes

Many other people have complained about XML unfriendly syntax too, and many have proposed alternatives. Simply search "XML alternatives" of the Web and you'll find plenty! (One of which was actually called SML too! No resemblance to this one).

A few important ones are:

- ASN.1 XER (XML Encoding Rules) [1] - ASN.1 is widely used in the telecom industry. XER is ASN.1 converted to XML.
  Pro: Powerful. XER documents compatible with XML document model.
  Con: Complex. Simpler alternatives now widespread.
- JSON JavaScript Object Notation [4] - The most popular of the alternatives now, by far.
  Pro: Powerful and simple. Easy to use, with I/O libraries available for most languages.
  Con: Not adapted for mixed content cases.
- Google Protocol Buffers [8] - Used internally by Google for all structured data transfers.
  Pro: Simple syntax. Compiler for generating compact and fast binary encodings for wire transfers.
  Con: Even Google seems to prefer JSON for public end-user APIs.

And *many* other proposals [2], with varying levels of success. Old and new examples:

- YAML Ain't Markup Language [21] - A human readable serialization language, inspired by Internet Mail syntax.
- {mark} [6] - A JSON+XML synthesis, announced in Jan. 2018.
  Pro: A simple and very readable syntax. All JSON and XML features, allowing to replace either without missing anything.
  Con: Incompatible with both.

1.1.2. Subsets of XML

Others have also attempted to "fix" XML by keeping only a subset of XML. The W3C themselves have made a such a proposal, called Simple XML [9]. The Wikipedia page for that same (?) proposal [10]) goes much further, by abandoning attributes. Although this does make the tree structure simpler, this definitely does not make the document more readable. MicroXML [7] discussed further down is also in this category, abandoning declarations and processing instructions.

1.2. Alternative representations of XML

1.2.1. Binary representation

Several groups have proposed binary representations of XML, including one that has been officially endorsed by the W3C:

- Efficient XML Interchange (EXI) Format 1.0 [3]

These methods address a different problem, which is finding the smallest and most efficient way to transfer XML data. Yet they prove one thing, which is that alternative representations of XML are possible and practical.

1.2.2. JSON representation

- MicroXML [7] - A subset of XML, that can be presented using a JSON syntax.
  Pro: Brings attributes to standard JSON.
  Con: The JSON version is longer than both SML and XML. No declarations nor processing instructions.
- The XSLT xml-to-json function [19] is part of a scheme allowing to convert JSON to a subset of XML, and that XML back to JSON. But it cannot convert any XML, only an XML representation of JSON.
  That XML-to-JSON back conversion can also be done using an XSLT style sheet.

This XSLT json-to-xml and xml-to-json scheme is basically the inverse of MicroXML:

Table 1.

MicroXML	XML → JSON representation of XML → XML
XSLT scheme	JSON → XML representation of JSON → JSON

Yet neither proposal can ensure full compatibility between JSON and XML.

1.3. Birth of the SML concept

At the same time I had these problems with the XML configuration files for Heartbeat, I was writing Tcl scripts for managing Lustre file systems on that cluster. The instances of my scripts on every node were exchanging increasingly big Tcl structures (As strings, embedded in network packets), for synchronizing their action. And I kept finding this both convenient, and easy to program and debug. (i.e. Review the structures exchanged when something went wrong!)

And then I began to think that the two problems were linked: XML is nothing more than a textual presentation of a structured tree of data. A Tcl program or a Tcl data structure is also a textual presentation of a structured tree of data. And the essence of XML is not its <tagged><blocks>, but rather its logical structure with a tree of elements, attributes, and content blocks with other embedded elements inside. In other words its DOM (Document Object Model).

All programs written in C, Java, Tcl, PHP, etc, share a common simple syntax for representing program trees {based on {nested blocks} surrounded by parentheses}, which is much easier to read by humans than the <tagged><blocks> used by XML</blocks></tagged>. The Tcl language has the simplest syntax in that family, with a grammar with just a dozen rules, and punctuation marks optional in simple cases. This makes it particularly easy to read and parse. And its one-instruction-per-line standard (Like Python or Go) is a natural match to all canonically formatted XML data files with one element per line.

Instead of reinventing a new data structure presentation language, it should be possible to convert XML into an equivalent Tcl-like format, while preserving all the elements, attributes, and data structures.

This defined a new problem: Find a text format inspired by Tcl, which is simpler than XML, yet is strictly equivalent to it. Equivalent in the mathematical sense that any XML file can be converted to that simpler format, then back into XML with no change whatsoever.

Non-goals: Do not try to generate valid Tcl syntax at all. The result is actually incompatible with Tcl in general.

1.4. The SML Solution

Keep the XML DOM tree model with elements made of a tag, optional attributes, and an optional data block, but use a simpler text representation based on the syntax of the C family languages.

The basic idea is that XML and SML elements correspond to each other like this:

- XML elements: <tag attribute="value" ...>contents</tag>
- SML elements: tag attribute="value" ... {contents}

But the devil lies in the details, and it took a while to find a set of rules that would cover all XML syntax cases, allow fully reversible conversions, optimize the readability of real-world files, and remain reasonably simple. After experimenting with a number of alternatives, I arrived at the set of rules defined further down, which give good results on real-world documents.

Example 1. Example extracted from a Google Earth file:

XML (from a Google Earth .kml file)	**SML (generated by the sml script)**
`<?xml version="1.0" encoding="UTF-8"?>`	`?xml version="1.0" encoding="UTF-8"`
`<kml>`	`kml {`
`  <Folder>`	`  Folder {`
`    <name>Sites in the Alps</name>`	`    name "Sites in the Alps"`
`    <open>1</open>`	`    open 1`
`    <Folder>`	`    Folder {`
`      <name>Drome</name>`	`      name Drome`
`      <visibility>0</visibility>`	`      visibility 0`
`      <Placemark>`	`      Placemark {`
`        <description>Take off</description>`	`        description "Take off"`
`        <name>Mont Rachas</name>`	`        name "Mont Rachas"`
`        <LookAt>`	`        LookAt {`
`          <longitude>5.0116666667</longitude>`	`          longitude 5.0116666667`
`          <latitude>44.8355</latitude>`	`          latitude 44.8355`
`          <range>4000</range>`	`          range 4000`
`          <tilt>45</tilt>`	`          tilt 45`
`          <heading>0</heading>`	`          heading 0`
`        </LookAt>`	`        }`
`      </Placemark>`	`      }`
`    </Folder>`	`    }`
`  </Folder>`	`  }`
`</kml>`	`}`

The difference in readability should be immediately obvious!

Example 2. Another example in XSLT:

XSLT (from the XSLT 3.0 spec)	**SML (generated by the sml script)**
`<xsl:stylesheet`	`xsl:stylesheet\`
`  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`	`  xmlns:xsl="http://www.w3.org/1999/XSL/►`
`  version="3.0"`	`Transform"\`
`  expand-text="yes">`	`  version="3.0"\`
	`  expand-text="yes" {`
` <xsl:strip-space elements="PERSONAE"/>`	
	` xsl:strip-space elements="PERSONAE"`
` <xsl:template match="PERSONAE">`	
`   <html>`	` xsl:template match="PERSONAE" {`
`     <head>`	`   html {`
`       <title>The Cast of {@PLAY}</title>`	`     head {`
`     </head>`	`       title "The Cast of {@PLAY}"`
`     <body>`	`     }`
`       <xsl:apply-templates/>`	`     body {`
`     </body>`	`       xsl:apply-templates`
`   </html>`	`     }`
` </xsl:template>`	`   }`
	` }`
` <xsl:template match="TITLE">`	
`   <h1>{.}</h1>`	` xsl:template match="TITLE" {`
` </xsl:template>`	`   h1 "{.}"`
	` }`
` <xsl:template match="PERSONA">`	
`   <p><b>{.}</b></p>`	` xsl:template match="PERSONA" {`
` </xsl:template>`	`   p {b "{.}"}`
	` }`
`</xsl:stylesheet>`	
	`}`

Example 3. Another example in XML Schema:

Union datatype examp. (from the 1.1 spec)	**SML (generated by the sml script)**
`<attributeGroup name="occurs">`	`attributeGroup name="occurs" {`
`  <attribute name="minOccurs"`	`  attribute name="minOccurs"\`
`      type="nonNegativeInteger"`	`      type="nonNegativeInteger"\`
`      use="optional" default="1"/>`	`      use="optional" default="1"`
`  <attribute name="maxOccurs"`	`  attribute name="maxOccurs"\`
`      use="optional" default="1">`	`      use="optional" default="1" {`
`    <simpleType>`	`    simpleType {`
`      <union>`	`      union {`
`    <simpleType>`	`    simpleType {`
`      <restriction base='nonNegativeInteger'/>`	`      restriction base='nonNegativeInteger'`
`    </simpleType>`	`    }`
`    <simpleType>`	`    simpleType {`
`      <restriction base='string'>`	`      restriction base='string' {`
`        <enumeration value='unbounded'/>`	`        enumeration value='unbounded'`
`      </restriction>`	`      }`
`    </simpleType>`	`    }`
`      </union>`	`      }`
`    </simpleType>`	`    }`
`  </attribute>`	`  }`
`</attributeGroup>`	`}`

2. SML Syntax rules

(Note: This is not a BNF grammar, but rather a list of principles, that allow to successfully convert XML ↔ SML.)

2.1. Elements

- Elements normally end at the end of the line.
- They continue on the next line if there's a trailing '\'.
- They also continue if there's an open "quotes" or {curly braces} block.
- Multiple elements on the same line must be separated by a ';'.

2.2. Attributes

- The syntax for attributes is the same as for XML. Including the rules for using quotes and escape chars. (And so is different from SML's text elements quoting syntax, which allows quoting any text with ' & ".)
- There must be at least one space between the last attribute and the beginning of the content data.

2.3. Content data

- The content data are normally inside a {curly braces} block.
- The content text is between "quotes". Escape '\' and '"' with a '\'.

- If there are no further child elements embedded in contents (i.e. it's only text), the braces can be omitted.
- Furthermore, if the text does not contain blanks, '"', '=', ';', '#', '{', '}', '<', '>', nor a trailing '\', the quotes around the text can be omitted too. (i.e. If the text cannot be confused with an attribute or a comment or any kind of SML markup.)

2.4. Other types of markup

All use the same rules as the elements for juxtaposition and continuation.

- This is a **?Processing instruction** . (The final '?' in XML is removed in SML.)
- This is a **!Declaration** . (Ex: a !doctype definition)
- This is a #**-- Comment block, ending with two dashes --** .
- Simplified case for a # **One-line comment** .
- This is a **<[[Cdata section]]>** . An optional new line, immediately following the opening <[[, is discarded if present.
- Non significant spaces in closing tags are noted with a second set of curly braces immediately following the content data block. Ex: tag {content}{ }.

2.5. Heuristics for XML↔SML conversion

- Spaces/tabs/new lines are preserved.
- The sml program adds one space after the end of the element definition (i.e. after the last attribute and optional trailing spaces inside the element head), before the beginning of the data block. This considerably improves the readability of the sml output. Then it removes it when converting SML back to XML. An SML file is invalid without that space anyway.
- Empty data blocks (i.e. Blocks containing just spaces) encoding: Use {} for multi-line blocks, and "" for single-line ones.
- Unquoted attribute values are accepted, in an attempt to be compatible with HTML-style attributes, which do occur in poorly-written XML files.

2.6. Syntax rules discussion

XML files without mixed data usually contain a hierarchy of outer elements embedded within each other with no text. Then the terminal elements (the innermost elements) contain just text.

- *SML elements normally end at the end of the line*. A natural match for canonically formatted XML files, with one XML terminal element per line.
- *They continue on the next line if there's a trailing '\'*. Same rule as for Tcl, and many other programming languages.

- *They also continue if there's an open "quotes" or {curly braces} block.* This is a major advantage of the Tcl syntax, allowing to minimize the syntactic glue characters.
- *Multiple elements on the same line must be separated by a ';'.* Again, the same as Tcl.
- *The syntax for attributes is the same as for XML:* `name="value"` with value between 'single' or "double" quotes, and using references (like &amp; , &lt; , &gt; , &apos; , &quot;) to escape the special characters in values. I considered using Tcl's quoting rules instead. But this made the conversion program more complex, and did not make the SML more readable. (Actually it made it less readable, making it more difficult to read long lists of attributes.) Most real-world attribute values will look exactly the same as the equivalent Tcl string anyway. TDL [13] proposes an interesting alternative: Write attributes as functions named options, with a dash: `-name value` Pro: Easier to parse in Tcl. Con: Less intuitive to people who don't know Tcl. Con: Makes it more difficult to deal with HTML-like attributes that have no value.
- *The content data are normally inside a {curly braces} block. Braces in the content text must be escaped by a '\'.* Same as Tcl {blocks}. Works well for XML outer elements containing inner elements.
- *If there are no further child elements embedded in contents (i.e. only text), the braces can be omitted.* A major readability improvement. The quoting rules for the text ensure that the text content cannot be confused with an additional attribute.
- *The quotes around text can be omitted if the text does not contain blanks, '"', '=', ';', '#', '{', '}', '<', '>', nor a trailing '\', and if there are no other elements at the same tree depth. (i.e. It cannot be confused with an attribute or a comment or any kind of SML markup.)* Maximizes readability by removing all extra characters around simple values. Possible alternative: In the cases where text and elements are mixed at the same tree depth (Like in XHTML, DocBook, etc), use a pseudo element tag like !text or just @ (But not #text which would look like a comment) to flag it. This would allow extending the SML syntax to support element names with spaces. See the "show script" section below for a useful application of that.
- *This is a* ?Processing instruction . *This is a* !Declaration . (Ex: A !doctype definition) Both are treated like XML empty elements, with a name beginning with an '?' or a '!'. All contents are preserved, except for the final ?> and > respectively. Add a '\' at the end of lines if the element continues on the following lines.
- *Simplified case for a* # One-line comment . Same as for Tcl, and many other scripting languages.

- *This is a* #-- Comment block -- . I considered using other syntaxes, like <# Multi-line comment #> in PowerShell. But this was barely more concise, and this created problems to deal with the -- sequence in SML (not valid in an XML comment), or the #> sequence in XML (not valid in an SML comment in that case) In fine, the simplest was to stick to the -- delimiters like in XML.
- *This is a* <[[CDATA section]]> Like for comment blocks, sticking to the XML termination sequence proved to be the easiest option. Any other type of delimiter would have required complex escaping rules, in case that delimiter appears in the CDATA itself. The possibility of having adjacent CDATA sections would have made these rules even more complex. By symmetry, I used `<[[` for the opening sequence. Note that the CDATA`]]>` end markers cannot be confused with the `]]>` end markers at the end of some complex !declarations, because those ones become `]]` after the final '>' is removed in SML. *An optional new line, immediately following the opening <[[, is discarded.* This makes it easy to view multiple lines of CDATA. The first line will begin on the first column, like all the others. Gotcha: That additional new line <u>must</u> be inserted if the CDATA begins with an initial new line. Else the initial new line would be lost during the conversion back to XML. Possible alternative: I experimented with simpler alternatives in other programs. One is the indented block, used in the show.tcl script described further down:

  ```
  Preceding content{
    This is a sample CDATA with an XML <tag>
  }Following content
  ```

  Here, the rule is that all CDATA block contents are indented by two more spaces than the previous line. The first '}' at the same indentation as the opening '{' sign marks the end the CDATA. The CDATA begins after the new line following the opening '{' (So this new line is not optional here), and ends before the final new line before the closing '}'. Pro: More lightweight syntax, more in the spirit of Tcl. Pro: Looks better in deep trees, as multi-line CDATA blocks are indented like the rest. Con: Adds numerous spaces, and makes the CDATA block weight more in bytes. Con: Made the sml conversion program more complex and slower. Variation on the same theme: Particular case of a CDATA section that makes up the whole content of an element: Instead of encoding this content block with double parenthesis `{{\n CDATA\n}}`, it'd be written `={\n CDATA\n}`

3. SML characteristics

3.1. SML files size

An interesting side benefit of the conversion is that the total size of the converted files is 12% smaller than the original XML files. (Tested on a 1MB set of real files gathered at work.) Among big files, that reduction goes from 4% for a file with lots of large CDATA elements, to 17% for a file with deeply nested elements.

Even after zipping the two full sets of samples, the SML files archive is 2% smaller than the XML files archive. Not much I admit, but this would help Microsoft alleviate the Office documents bloat. ☺

As for XML compression, many dedicated compressors are available (Ex: [14], [18]). Obviously they give better results than SML. But just as obviously the compressed files are unreadable by humans!

Reductions are much better on xml documents using name spaces. For example on the sample SOAP envelope from the SOAP 1.2 specification, the gain is 30%. Transporting SOAP messages in their SML form instead of XML would yield huge network bandwidth gains! (In case somebody wants to revive SOAP! ☺)

3.2. Effect on mixed content

As mentioned already, mixed content files can be successfully converted to SML and back. But when there's a mix of text and markup on the same line the SML version is not much simpler to read than the XML one.

Example 4. In a simple XHTML example…

Formatted text	A line of text with **bold and *bold+italic*** parts.	Size
XHTML	<p>A line of text with <b>bold and <i>bold+italic</i></b> parts.</p>	68
SML	p {"A line of text with"; b {"bold and"; i bold+italic}; "parts"}	65

… the SML version is indeed a bit shorter. Yet I find it already more difficult to understand than the original XML.

Example 5. But with a little more complex text and formatting …

Formatted text	By definition, "**1mm = 1000µm.**"	Size
XHTML	<p style="color:blue">By definition, "<b>1mm = 1000&micro;m.</b>"</p>	69
SML	p style="color:blue" {"By definition, \\"";b "1mm = 1000&micro;m.";"\\""}	71

… the SML size is actually longer (71 characters instead of 69 for the XML), and the SML quoting rules become confusing, to the point of making it hard for humans to distinguish the text, markup, and attributes.

With even more complex mixed content XML, the tendency continues, and SML becomes ever bigger and harder to read for humans.

On the other hand, when the mixed content is formatted and indented as canonic XML (with at most one element per line), then the conversion yields relatively simple SML, with a significantly smaller size. For example, at some stage, this very article was saved as a 64,309 bytes DocBook XML file. Then sml.tcl could convert this XML to a 59,422 bytes SML file, still very agreeable to read.

3.3. Comparison with other data serialization formats

(Note: The two columns may overflow when printed. Best viewed on screen as HTML.)

3.3.1. SML versus XML

SML	XML
`root {`	`<root>`
`  # One-line comment`	`  <!-- One-line comment -->`
`  #-- Long comment`	`  <!-- Long comment`
`       spanning 2 lines --`	`       spanning 2 lines -->`
`  empty`	`  <empty/>`
`  number type="real" 3.14`	`  <number type="real">3.14</number>`
`  word yes`	`  <word>yes</word>`
`  sentence "Hello XML world"`	`  <sentence>Hello XML world</sentence>`
`  sub1 {"with mixed text"`	`  <sub1>with mixed text`
`    sub2 "and inner elements"`	`    <sub2>and inner elements</sub2>`
`    "and" ;sub3; ;sub4 more`	`    and <sub3/> <sub4>more</sub4>`
`  }`	`  </sub1>`
`  <[[ SML <==> XML ]]>`	`  <![CDATA[ SML <==> XML ]]>`
`}`	`</root>`

3.3.2. SML versus MicroXML presented as JSON

SML	MicroXML presented as JSON
`root {`	`["root", {}, [`
`  # One-line comment`	
`  #-- Long comment`	`  (Note: There are no comments in JSON)`
`       spanning 2 lines --`	
`  empty`	`  ["empty", {}, []],`
`  number type="real" 3.14`	`  ["number", {"type":"real"}, ["3.14"]],`
`  word yes`	`  ["word", {}, ["yes"]],`
`  sentence "Hello XML world"`	`  ["sentence", {}, ["Hello XML world"]],`
`  sub1 {"with mixed text"`	`  ["sub1", {}, ["with mixed text",`
`    sub2 "and inner elements"`	`    ["sub2", {}, ["and inner elements"]],`
`    "and" ;sub3; ;sub4 more`	`    "and", ["sub3", {}, []], ["sub4", ►`
`  }`	`{}, ["more"]]`
`  <[[ SML <==> XML ]]>`	`  ]],`
`}`	`  " SML <==> XML "`
	`]]`

3.3.3. SML versus {mark}

SML	{mark}
`root {`	`{root`
`  # One-line comment`	`  // One-line comment`
`  #-- Long comment`	`  /* Long comment`
`       spanning 2 lines --`	`       spanning 2 lines */`
`  empty`	`  {empty}`
`  number type="real" 3.14`	`  {number type:"real" 3.14}`
`  word yes`	`  {word "yes"}`
`  sentence "Hello XML world"`	`  {sentence "Hello XML world"}`
`  sub1 {"with mixed text"`	`  {sub1 "with mixed text"`
`    sub2 "and inner elements"`	`    {sub2 "and inner elements"}`
`    "and" ;sub3; ;sub4 more`	`    "and" {sub3} {sub4 "more"}`
`  }`	`  }`
`  <[[ SML <==> XML ]]>`	`  " SML <==> XML "`
`}`	`}`

4. The sml.tcl conversion script

4.1. Presentation

A well tested XML↔SML conversion program, called **sml.tcl**, is open-sourced and available at the URL: https://github.com/JFLarvoire/SysToolsLib/blob/master/Tcl/sml.tcl

It works in any system with a Tcl interpreter. (Standard in Linux: Just rename the script as **sml** and make it executable. In Windows, a free Tcl interpreter is available at http://www.activestate.com/activetcl ; For recommendations on how to best configure it, see https://github.com/JFLarvoire/SysToolsLib/tree/master/Tcl.)

It is able to convert any XML file to SML, then back into XML, with the final XML files binary equal to the originals. The script is usable in a pipe. It auto-detects if the input is XML or SML, and outputs the other representation. Use `sml -?` or `sml -h` to display the help screen.

A simple glance at the contents of the SML files will show, as in the Google Earth example above, that the "useful" information is much easier to find. The eye is not distracted anymore by the noise of useless end tags and brackets.

4.2. Test methodology

I've first tested it on a large number of sample XML files from various sources at work, totaling about 1 MB.

And of course I've been using it regularly for several years.

More recently, I've tested it successfully with all the libxml2 (http://xmlsoft.org/) test cases. The only exceptions are the test files encoded in exotic (for me) text encodings like EBCDIC or UTF-16. This is a limitation of the sml.tcl script, but in no way a limitation of the SML syntax. The script works fine with ASCII and UTF-8, and I don't plan to add support for anything else.

In both cases the testing relies on a self-test routine in the script, triggered by using the `sml -t` option.

`sml -t` converts all files of types {*.xml *.xhtml *.xsl *.xsd *.xaml *.kml *.gml} in the current directory to sml, then converts the sml file to xml, then compares each final xml file to the initial one. Any problem during one of the conversions, or if the final file does not match byte-for-byte the initial one, is reported. And in the end it displays statistics about the number of files tested, etc.

There's an option to change the list of file types to test, if desired.

`sml -t -r` does the same recursively in all subdirectories.

4.3. Performance

The file has about 3000 lines of code, half of which are an independent debugging library.

The only issue is performance: It converts about 10 KB/s of data on a 2 GHz machine. This is perfectly fine for small XML files, but can be cumbersome with very large files. Rewriting it in C and optimizing the lowest I/O routines should be able to increase performance by orders of magnitude. I've begun to do that with the libxml2 library.

4.4. Known limitations

- As explained above, only ASCII (+ 8-bit supersets) and UTF8 text encodings are supported now.
- The converted files use the local operating system line endings (\n or \r or \r \n). So if the initial XML file was encoded with line endings for another operating system, converting it to SML then back will not be binary equal to the initial file. But it will still be logically equal, as the XML spec states that all line endings are equivalent to \n.

5. Support for SML in the libxml2 library

5.1. Presentation

I started work on a fork of the libxml2 library that can parse both XML and SML, and optionally output SML.

This fork is available on GitHub at https://github.com/JFLarvoire/libxml2.

Note that this is still a demonstrator with limited capabilities:

- It can parse well formed SML, but not yet declarations, processing instructions, etc. (Hopefully done by February)
- It can save DOM trees as SML. But it cannot yet write SML directly using the write APIs. Nor can it save HTML documents as SML.
- I have not tested any of the SAX APIs, so they probably do not work for SML.
- Of course all XML parsing, processing, and output capabilities are unchanged.
- A program called x2s.c reads either XML or SML, and outputs the other one.

Thanks to the equivalence between XML and SML, the changes are very small relative to the (huge) size of the library. Also note that half of the changes are actually debug instrumentation, which do not need to be retained in the final version.

Preliminary results show that x2s.exe is about 20 times faster than sml.tcl for converting large XML files to SML.

5.2. Non binary-reversibility

One noticeable result is that x2s.exe *cannot* convert XML files to SML, then back to XML, and yield files that are binary identical to the original one in all cases like sml.tcl does. This is due to a limitation of the libxml2 design, which does not record non-significant white spaces in markup. To allow binary compatibility, we'd need to add an option to parse a new kind of DOM node, recording that kind of non-significant spaces.

5.3. Issues with the xmlWriter APIs

I've started work on the xmlWriter module, and found one limitation: It will not always generate optimal SML (that is remove the {} or "" when possible) due to limitations of the current API. The reason is that the write APIs separate the opening of an element, the generation of its content, and the closing of the element. (Except for the special case of an empty element.) This does not allow to know when an element is opened if it'll contain just text (allowing to avoid using {}), or sub-elements (requiring the use of {}).

I see two ways to work around that limitation (actually not mutually exclusive):

- Add a new API function xmlTextWriterWriteElementAndItsText (+the Format and VFormat variants) Advantage: This would be usable with both XML and SML, and fix common cases. Drawback: This would still not fix the case of elements having attributes, etc. We'd need many new functions to cover all cases.
- Cache every new element in a temporary DOM sub-tree, then once complete, write that sub-tree. Advantage: This fixes all cases without requiring any change to the write API. Drawback: We lose the performance advantage of the write APIs.

6. Other scripts

6.1. The show script

This script allows serializing a whole file system tree as SML (And thus indirectly as XML).

Open-sourced and available at: https://github.com/JFLarvoire/SysToolsLib/blob/master/Tcl/show.tcl

The principle is that each file or directory is an SML element. Directories contain inner elements that represent files and subdirectories. File contents are displayed as text if possible, else are dumped in hexadecimal.

It also has options for generating several alternative experimental SML formats, which have helped convince me which was the most readable solution.

The show script has two major modes of operation:

- A simplified mode, which is not fully SML-compatible, but produces the shortest output, easiest to read. (This is the default mode of operation)

  This mode is particularly convenient for reviewing the content of Linux virtual file systems, like `/proc/fs`.
- A strict mode, which produces a fully SML-compatible output, at the cost of a heavier output.

The textual output can be (in theory) used to recreate the complete file system.

6.2. The spath script

This script does not exist, but this section is a thought experiment that gives some insight on the power of the SML concept.

Think of this as the reverse of the previous section: show.tcl was showing a file system as an XML text tree; here we're going to manage an SML or XML text tree as a file system.

I had made another script called xpath.tcl[3], which makes it easy to use XPATH to view the contents of XML files, or extract data from them. This script does nothing fancy. All it does is to pretend the XML file represents a file system, and allow accessing its contents using Unix-style commands like cat or ls. XML elements are considered as directories, and attributes as files. The content data for a terminal element is considered as an unnamed file. Examples:

`xpath sites.kml ls /kml/Folder/Folder`

lists all inner elements as directories, and attributes as files.

`xpath sites.kml cat /kml/Folder/Folder/name`

Displays attribute values, or the text content for elements. Here it outputs "Drome".

The idea here is to write an spath.tcl script that does the same for SML data instead of XML.

Supporting all features of XPATH would be difficult, as xpath.tcl uses Tcl's TclDOM package to do the real work with XPATH transforms. But in the short term, it's possible to get the same functionality using a one-line spath shell script:

`sml | xpath %*` (%* for Windows cmd, or $* for Unix bash)

1) This example shows the power of having a data format that is equivalent to XML.

2) Notice how this works nicely with the output of the show.tcl script above *running in simplified mode*: show.tcl captures the contents of a real file system, where files are normally displayed with the `cat PATHNAME` command. Then spath allows extracting the contents of individual files from that SML file using `spath cat PATHNAME`. The `PATHNAME` is the same. Gotcha: Unfortunately this does not work with file names that are not XML tag compliant, for example if they contain spaces, or begin with a digit, etc. A possible addition to XML 2.0 maybe? ☺

7. Next Steps

- Call to action: Download the tools, and try with them with your XML data. Please send me (with [SML] in the email subject) feedback about the SML syn-

[3] https://github.com/JFLarvoire/SysToolsLib/blob/master/Tcl/xpath.tcl

tax, and the possible alternatives. Is there any error or inconsistency that remains, preventing full XML compatibility in some case? And please report any problem with the tools themselves as issues in their respective GitHub area.

- Continue work to improve SML parsing and generation as an option to the libxml2 library, or any other similar XML management library. Anybody interested in participating?
- If interest grows, work with interested people to freeze a standard.
- Any project which stores data as XML files, even zipped like in MS Office, will save space and increase ease of use by using the SML format instead. What about yours?
- The savings potential is even better in XML-based network protocols, such as SOAP. Adapting existing XML-based protocols to use SML instead is easy, and will significantly increase bandwidth. Creating new ad hoc SML-based protocols would be easy too, and packet analysis would be much easier!
- Any new project which does not know what data format to use, could get an easy-to-use format by adopting this SML format, while ensuring compatibility with XML-compatible-only tools, should the need arise.

Bibliography

[1] ITU *XML encoding rules (XER) for ASN.1*: `http://asn1.elibel.tm.fr/xml/xer.htm`

[2] Wikipedia *Comparison of data serialization formats*: `https://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats`

[3] W3C *Efficient XML Interchange (EXI) Format 1.0* specification: `https://www.w3.org/TR/2014/REC-exi-20140211`

[4] *Introducing JSON* (JavaScript Object Notation): `https://www.json.org/`, and ECMA *The JSON Data Interchange Syntax*: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

[5] J.F. Larvoire *libxml2 fork supporting SML* XML↔SML conversion script: `https://github.com/JFLarvoire/libxml2`

[6] Henry Luo *{mark}* presentation: `https://mark.js.org/`

[7] W3C *MicroXML Community Group*: `https://www.w3.org/community/microxml/`

[8] Google *Protocol Buffers*: `https://developers.google.com/protocol-buffers/`, and Google Open Source Blog: `http://google-opensource.blogspot.fr/2008/07/protocol-buffers-googles-data.html`

[9] W3C *Simple XML*: `http://www.w3.org/XML/simple-XML.html`

[10] Wikipedia *Simple XML*: `http://en.wikipedia.org/wiki/Simple_XML` (Apparently unrelated to the previous one, despite the link)

[11] J.F. Larvoire *sml.tcl* XML↔SML conversion script: `https://github.com/JFLarvoire/SysToolsLib/blob/master/Tcl/sml.tcl`

[12] Tcl wiki *XML links page*: `http://wiki.tcl.tk/1740`

[13] Tcl wiki - Lars Hellström *TDL proposal*: `http://wiki.tcl.tk/25681`

[14] Open Mobile Alliance WBXML - Wireless*Binary XML Content Format Specification*: `http://www.openmobilealliance.org/tech/affiliates/wap/wap-192-wbxml-20010725-a.pdf`

[15] W3C *Extensible Markup Language (XML) 1.0* specification: `http://www.w3.org/TR/xml/`

[16] Paul T *A list of XML alternatives proposals*: `http://www.pault.com/xmlalternatives.html` (Dead link), and *On Data Languages*: `http://www.pault.com/data-languages.html`

[17] James Cheney *XML compression bibliography*: `http://xmlppm.sourceforge.net/paper/node9.html`

[18] James Cheney *Compressing XML with Multiplexed Hierarchical PPM Models*: `http://xmlppm.sourceforge.net/paper/paper.html`

[19] W3C XSLT *xml-to-json* function: `https://www.w3.org/TR/xslt-30/#func-xml-to-json`

[20] Tcl wiki *xmlgen* presentation: `http://wiki.tcl.tk/5976?redir=3210`

[21] yaml.org *YAML Ain't Markup Language*: `http://yaml.org/`

Can we create a real world rich Internet application using Saxon-JS?

Pieter Masereeuw
<pieter@masereeuw.nl>

Abstract

This paper describes how Saxon-JS[1] techniques can be used in creating a rich internet application as a front end for dictionary lookup. The main purpose of the text is to demonstrate, by means of some examples, how easy it is to avoid JavaScript, almost entirely, in favour of Saxon-JS' implementation of XSLT - in the author's humble opinion being the web's best programming language. Apart from that, it will discuss some difficulties that were encountered while writing the application, plus their solutions - if any.

1. The challenge

In its long history, the Institute for the Dutch language (*Instituut voor de Nederlandse taal - INT*) has created many scientific dictionaries describing Dutch vocabulary, many of them of impressive size, as can be seen in the table below.

Table 1. Dictionaries made available in the application created by the INT

Dictionary	Date range	# Entries	# Citations
Dictionary of old Dutch (oudnederlands woordenboek)	A.D. 500 - A.D. 1200	8954	≈ 30000
Dictionary of early medieval Dutch (vroegmiddelnederlands woordenboek)	A.D. 1200 - A.D. 1300	≈ 25000	≈ 26000
Dictionary of medieval Dutch (middelnederlands woordenboek)	A.D. 1250 - A.D. 1550	≈ 75000	≈ 400000
Dictionary of the Dutch language (woordenboek der Nederlandse taal)[a]	A.D. 1500 - A.D. 1976	≈ 400000	≈ 1700000
Dictionary of the Frisian language (wurdboek fan de Fryske taal)[b]	A.D. 1800 - A.D. 1975	≈ 118000	n/a

[a]Being the world's largest dictionary, the printed edition consists of 43 volumes having a total length of three meters. Work on it took 134 years.

[b]The printed version consists of 25 volumes.

[1]http://www.saxonica.com/saxon-js/index.xml

Figure 1. Just a few volumes of the world's largest dictionary

Originally being printed books, these dictionaries are nowadays also available as a web application.

The dictionaries can all be consulted simultaneously by means of an internet application, called *GTB* (*geïntegreerde taalbank*, integrated language bank). Unfortunately, this application was developed using Flash[2]. For obvious reasons, a new application had to be created and it was decided to jump ahead of current JavaScript-based solutions in order to see whether XSLT could serve as the web's programming language of the future.

2. About Saxon-JS

The use of XSLT in the browser has been discussed by Debbie Lockett and Michael Kay at the 2016 Balisage conference[3].

Saxon-JS was designed to be an XSLT 3.0 implementation that can be used as a programming language for creating interactive web pages. It can do many things without JavaScript, but both languages can also live in perfect harmony and use of each other's benefits. Also, popular frameworks such as JQuery and those based on it, can stay in place, if necessary.

In order to use XSLT in the browser, you need to compile your stylesheet with Saxon-EE (the Enterprise Edition of the popular XSLT engine[4]). After that, the compiled stylesheet must be referenced from a web page, using a tiny piece of

[2]Generated by OpenLaszlo, which is a discontinued open source platform for the development and delivery of rich Internet applications (source: Wikipedia).

[3]Lockett, Debbie, and Michael Kay. "Saxon-JS: XSLT 3.0 in the Browser." Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016. In *Proceedings of Balisage: The Markup Conference 2016*. Balisage Series on Markup Technologies, vol. 17 (2016). DOI: 10.4242/BalisageVol17.Lockett01 (http://www.balisage.net/Proceedings/vol17/html/Lockett01/BalisageVol17-Lockett01.html).

[4]The Oxygen XML Editor comes with Saxon-EE and has a facility to easily compile stylesheets with it.

JSON. The Saxon-JS runtime itself is written in JavaScript and does not need a paid license.

3. About the application

The new GTB application uses the popular Bootstrap[5] library as basis of its front end design. One of the benefits of using Bootstrap is its excellent set of CSS definitions in combination with JQuery and normal JavaScript.

The application can be viewed here: http://gtb.ivdnt.org [6]. Using Bootstrap speeds up the development process by providing easy CSS and JavaScript facilities for a responsive website with a modern look. As a matter fact, in our case, Bootstrap is almost exclusively used for its CSS definitions and not for its JavaScript routines - this might have been possible, but our intention was to create a front end application that would minimize the use of JavaScript as much as possible.

4. Modifying the current document by event response

The normal way in which a rich client operates, is by modifying the HTML document that is presented in the browser. Typical actions are adding text, HTML elements and attributes.

Programmers who are familiar with XSLT may wonder how the language can support user interactions, such as mouse clicks and keyboard events. The answer is that Saxon-JS has extended XSLT template modes to correspond with JavaScript events.

For example, the JavaScript `onclick` event corresponds with the Saxon-JS mode `ixsl:onclick`.

Furthermore, instead of generating a new document, Saxon-JS is capable of modifying the current document in the browser window, by manipulating attributes and by adding content using an extended behaviour of the `xsl:result-document` construct.

5. Hiding and showing content

The following code snippet shows how to select a tab in response to a mouse event (in order to show search results). If a click (`mode="ixsl:onclick"`) occurs on the search button, it tracks the HTML `<div>` element that represents the tab and then adds, in the `do-search` template, the Bootstrap value `active` to the class

[5]Bootstrap is an open source toolkit for developing with HTML, CSS, and JavaScript; http://getbootstrap.com/.

[6]That should be the url at the time of the XML Prague 2018 conference. If, heaven forbid, the Flash application is still displayed, please refer to http://gtb.ivdnt.org/search/.

attribute of that `<div>`, but removes it from the sibling `<div>` elements (being the other tabs). Adding attributes is done by using the extension instruction `ixsl:set-attribute`.

Please note that the `ixsl` prefix refers to the Saxon-JS namespace and that identifiers with the `ivdnt` prefix have been defined elsewhere in the application[7]. Their definitions are not shown here for brevity's sake.

Example 1. ixsl:onclick and ixsl:set-attribute

```
<xsl:template match="button[@name eq 'do-search']" mode="ixsl:onclick">
    <xsl:variable name="tabdiv" as="element(div)" select=" … "/>

    <xsl:call-template name="ivdnt:do-search">
        <xsl:with-param name="tabdiv" select="$tabdiv"/>
    </xsl:call-template>
</xsl:template>

<xsl:template name="ivdnt:do-search">
    <xsl:param name="tabdiv" as="element(div)" required="yes"/>
    <!-- Deactivate all tabs but activate $tabdiv: -->
    <xsl:for-each select="$tabdiv/../div">
        <xsl:variable name="class-without-active" as="attribute(class)"
            select="ivdnt:remove-class-value(@class, 'active')"/>
        <ixsl:set-attribute name="class"
            select="ivdnt:add-class-values($class-without-active,
                       if (. is $tabdiv) then ('active') else ())"/>
    </xsl:for-each>

    <!-- Start the search: -->

    <!-- … code removed for brevity … -->
</xsl:template>
```

6. Interfacing with JavaScript

Although many operations can be done by means of the facilities offered by Saxon-JS, there are things that still need to be done the traditional way. In most cases, this concerns effects beyond the modification of the document shown in the browser window.

For this situation, it is possible to call JavaScript from XSLT.

Typically, an XPath expression in the special namespace for JavaScript (`js` prefix) makes it possible to call user-defined JavaScript functions. Alternatively, the `ixsl:call()` function can be used.

[7]The institute recently changed its preferred acronym to INT; the ivdnt XML prefixes show the earlier convention.

The following two snippets illustrate this; the function `ivdnt:always-false()` is a function that always returns `false()` by doing a silly comparison, thus preventing the output of `ixsl:call()` becoming part of the resulting sequence[8]. The third parameter passed to `ixsl:call()` is an array with parameters for the JavaScript `focus()` function. In this case, there are none.

Example 2. Calling a function for a JavaScript object

```
<xsl:sequence
        select="ixsl:call($textbox, 'focus', [])[ivdnt:always-false()]"/>
```

Example 3. Calling a user-defined JavaScript function

```
<xsl:sequence
        select="js:openNewWindow($url)[ivdnt:always-false()]"/>
```

7. Global storage - maintaining history

One of the features of the original Flash application was a facility to remember earlier queries made by the user. In order to accommodate this, some form of global browser storage is needed. Since users can use specific tabs for the queries, it will be convenient to associate the query history with each distinct tab.

Saxon-JS allows you to store anything as a property of a JavaScript object, and HTML DOM elements are such objects.

In order to remember the queries, the application will maintain an XML list structure containing XML representations of each submitted query. This list is stored as a JavaScript property at the `<div>` corresponding to the current tab.

Creating an XML structure for the submitted query is as simple as XSLT can be: just use `<xsl:apply-templates/>`, which will eventually transform HTML `<input>` and `<select>` elements into an XML representation. The following snippet shows this.

Example 4. A function to create a representation of HTML input elements

```
<xsl:function name="ivdnt:get-inputs-and-selects"
                   as="element(inputs-and-selects)">
    <xsl:param name="tabdiv" as="element(div)"/>
    <inputs-and-selects>
        <xsl:apply-templates
            select="$tabdiv//*[self::input | self::select]"
            mode="ivdnt:inputs-and-selects"/>
```

[8]This is needed in order to fool the optimizer of the Saxon compiler. The silly comparison checks if the current date equals the date on which the author was born (Saxon has no clue). Passing `false()` directly would cause the optimizer to forget about the instruction altogether, so no call would be made at all.

```
        </inputs-and-selects>
    </xsl:function>
```

In order to add a new representation to an existing list, the stored JavaScript object is retrieved using Saxon-JS' `ixsl:get()` function, by passing the JavaScript object/DOM element and the property name as parameters. After that, the new query representation is added to that list and stored using `<ixsl:set-property>`. This is exactly what the following named template does.

Example 5. Storing and retrieving global data

```
    <xsl:template name="ivdnt:store-inputs-and-selects">
        <xsl:param name="tabdiv" as="element(div)"/>


        <!-- Retrieve existing lists: -->
        <xsl:variable name="existing" as="element(inputs-and-selects)*"
            select="ixsl:get($tabdiv, 'prop-inputs-and-selects-list')/*"/>

        <!-- Calculate new list: -->
        <xsl:variable name="new"
            as="element(inputs-and-selects)"
            select="ivdnt:get-inputs-and-selects($tabdiv)"/>

        <!-- Join existing and new list: -->
        <xsl:variable name="new-list"
              as="element(inputs-and-selects-list)">
                <inputs-and-selects-list>
                    <xsl:copy-of select="$existing"/>
                    <xsl:copy-of select="$new"/>
                </inputs-and-selects-list>
        </xsl:variable>

        <!-- Store the result: -->
        <ixsl:set-property name="prop-inputs-and-selects-list"
            select="$new-list" object="$tabdiv"/>
    </xsl:template>
```

When the user requests the list of earlier queries, presenting such a list is, again, simply a matter of passing the stored list to `<xsl:apply-templates>`, this time in a special mode with the purpose of generating user-friendly query descriptions. Finally, the representations can be used to reset all user interface elements in the current tab to the values they had at the time of the selected query.

Obviously, manipulating global data spoils XSLT's feature of being a purely functional language without side effects. Below (Section 8.2) we will see some of the consequences.

8. Displaying search results

The most important facility of the application is of course to show the output of dictionary queries. As with all Internet applications, the application should not block the browser during a network operation; at the same time, a visual effect is required to inform the user that a lengthy operation is going on. And evidently, there must be a way to render the results (which are returned by the server in XML format) in the appropriate browser tab.

8.1. Preventing the application from freezing

Seeing that is was only intended to replace the GTB application's front end and leave the back end unmodified, the rest-like URL's of the old program had to be re-used. In order to prevent an application from freezing during network operations, Saxon-JS offers the `<ixsl:schedule-action>` extension, which schedules execution of a named template to take place as soon as the document pointed to by some URL has been loaded[9].

The following code fragment illustrates this.

Example 6. Asynchronous template call

```
<xsl:template name="ivdnt:show-results">
    <xsl:param name="url" as="xs:string" required="yes"/>
    <xsl:param name="tabdiv" as="element(div)" required="yes"/>

    <!-- Deactivate $tabdiv, show visual wait effects. -->

    <!-- … code removed for brevity … -->

    <ixsl:schedule-action document="{$url}">
        <xsl:call-template name="ivdnt:render-results">
            <!-- … parameters removed for brevity … -->
        </xsl:call-template>
    </ixsl:schedule-action>
</xsl:template>
```

After the XML search result has been loaded, it is converted to HTML in the normal XSLT way and inserted in a designated place in the HTML document of the

[9]An alternative way of using `<ixsl:schedule-action>` is to call a named template after a given delay, which can be useful to create animation.

application. For this, one can use the Saxon-JS extension of the `<xsl:result-document>` instruction by specifying a specific value for its `method` attribute and designating the target position by means of an id-reference, as seen in the next example[10].

Example 7. Showing the result document

```
<xsl:template name="ivdnt:render-results">
    <xsl:param name="url" as="xs:string" required="yes"/>
    <xsl:param name="tabdiv" as="xs:element(div)" required="yes"/>

    <xsl:result-document href="#resultarea" method="ixsl:replace-
content">
        <xsl:apply-templates select="doc($url)" mode="render-results"/>
    </xsl:result-document>

    <!-- Switch to result tab and hide visual wait effects. -->

    <!-- … code removed for brevity … -->
</xsl:template>
```

8.2. The order in which things happen

Since XSLT is, in its normal use, a language without side effects, evaluation of expressions can theoretically be done in any order, even parallelly. The normal, not-clientside, Saxon software, uses a technique called lazy evaluation[11], which means that the order of evaluation may be different from what is written in the stylesheet.

Now that XSLT side effects are in place, this proves to be somewhat cumbersome in some situations. In the preceding examples with the asynchronous template call, you may notice that things need to be done in a strict order:

1. Show a deactivation effect, such as a spinning icon.
2. Display the calculated result document.
3. Switch to the result tab and cancel the deactivation effect.

An early version of Saxon-JS applied the same lazy evaluation rules as its normal counterpart, spoiling the required visual effects. Fortunately, Saxonica was quick in fixing this issue.

[10]One may wonder whether the scheduled template starts running after the document has been loaded, or after all templates have been applied to it. The answer is probably the first. The current application is fast enough for this not to cause any problems, and furthermore, the user interface is only refreshed after the entire operation.
[11]Message by Michael Kay (Saxonica) on Saxonica's issue tracker for Saxon-JS.

9. Creating reusable components - an auto complete facility

One of the nice facilities of the original Flash application was *auto complete*. As the user typed in a textbox, suggestions were being presented on the basis of the dictionaries currently selected.

In the JavaScript world, adding such a facility is a matter of downloading a component of your choice and then modifying some call-back method to accommodate your own application rules.

Being a new technology, Saxon-JS evidently does not yet have such ready-to-deploy components. Therefore, it had to be written from scratch in XSLT, which proved to be less hard than anticipated. Preferably this should be a general purpose component, so an attempt was made to ensure that XSLT's `<xsl:import>` mechanism could be put to use for any specializations that may be needed[12].

10. When it is good to still have access to JavaScript

There are some situations where the use of JavaScript is still required. The situations that were encountered are:

- setting focus to some input element;
- calling third-party software, such as Google analytics;
- dealing with HTTP headers;
- global browser behaviour, such as opening a new window;

There is even one problem for which a solution has yet to be found:

- the option for users to cancel an asynchronous request (`<ixsl:schedule-action>`) if it takes too long.

11. Performance

Although no formal front end performance tests were carried out, it is safe to say that Saxon-JS performs very well. One of the most lengthy and complicated operations is the part that has to store or retrieve earlier queries (as mentioned above). This routine has to go over all the input elements in a given area in order to convert them to or from an internal XML data structure. In fact, even on a nine years old netbook, using this part of the software causes no observable delay. Therefore, we may conclude that Saxon-JS' performance seems to be adequate and very probably even better than that.

[12]One can wonder if the `<xsl:import>` mechanism is as powerful as JavaScript's call-backs. While the last operate on objects, `<xsl:import>` works more statically. Probably higher order functions can offer a more robust solution. For the moment, the current solution is powerful and easy enough.

12. Conclusions

The Institute for the Dutch language is quite happy with the new, non-Flash, version of its application. And so, the conclusion is: yes, we can create a real world rich Internet application using Saxon-JS.

The choice of a programming language depends, of course, on several factors. Personal preference plays an important role. When compared to JavaScript and frameworks that are based on it, XSLT and Saxon-JS offer the following benefits:

- Ease of use: although XML, and therefore XSLT, can be rather verbose, this is not necessarily a bad thing for a programming language. It is for this reason, for instance, that many people like Java better than Perl. In this respect, it seems safe to say that XSLT is better readable than JavaScript, which is often rather terse (albeit admittedly very powerful at the same time).
- Compiling a language, as done in the case of Saxon-JS, prevents many errors before a program is actually run.
- When reusing modules, name clashes are easily prevented by using namespaces.
- `<xsl:import>` provides a means for the specialization of standard components that is powerful enough for everyday use.

All-in-all, Saxon-JS makes XSLT in the browser a great success.

Implementing XForms using interactive XSLT 3.0

O'Neil Delpratt
Saxonica
<oneil@saxonica.com>

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Abstract

In this paper, we discuss our experiences in developing Saxon-Forms, a new partial XForms implementation for browsers using "interactive" XSLT 3.0, and suggest some benefits of this implementation over others. Firstly we describe the mechanics of the implementation - how XForms features such as actions are implemented using the interactive XSLT extensions available with Saxon-JS, to update form data in the (X)HTML page, and handle user input using event handling templates. Secondly we discuss how Saxon-Forms can be used, namely by integrating it into the client-side XSLT of a web application, and examples of the advantages of this architecture. As a motivation and use case we use Saxon-Forms in our in-house license tool application.

Keywords: XML, XSLT, XPath, XForms, Saxon, Saxon-JS

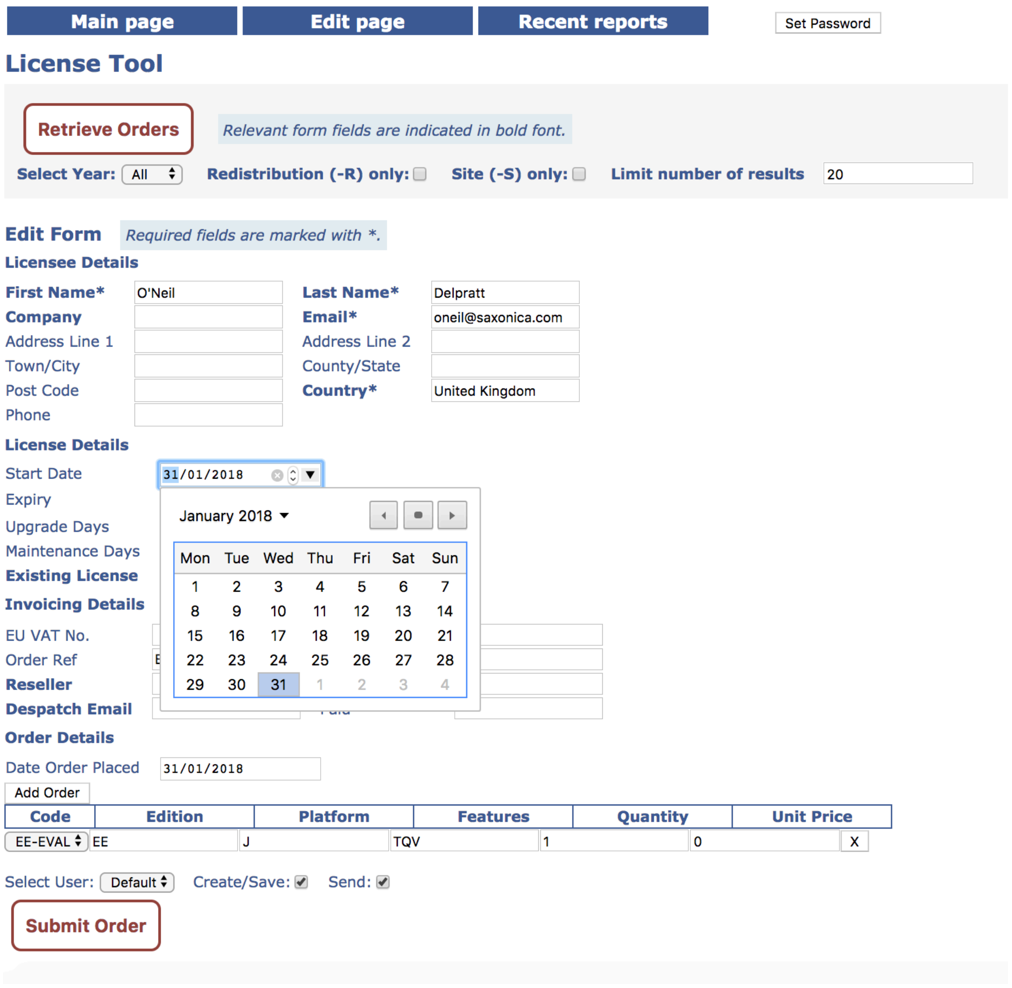
1. Introduction

1.1. Use-case: License Tool application

The motivation for developing Saxon-Forms was a specific use case - namely a project to improve our in-house license tool application (a form-based application for managing and generating licenses). The application used XForms [1] in the browser (using XSLTForms [2]) in the front-end, with server-side XSLT (and Java) processing in the back-end. The project was motivated first, by business needs to improve functionality in an in-house application that has slowly become unmaintainable, and secondly, by the fact that we wanted to improve the capability of Saxon-JS [3] to handle real-world applications with both front-end and back-end processing. We felt that using the technology for an in-house application would be the best way to discover what product enhancements were needed.

The license tool architecture redesign is discussed in detail in [4], where the focus is on the redistribution of XSLT processing, by using Saxon-JS in the browser for client-side XSLT. In this paper, our focus is another part of the project: the use of XForms. Rather than using existing implementations of XForms which run in the browser (such as XSLTForms), alongside the client-side of the application which is written in interactive XSLT [5] [6] and runs in Saxon-JS, we set out to work towards a new implementation of XForms 1.1 which would also run in Saxon-JS. This would allow us to better integrate the use of XForms into the client-side application, as well as being a further exercise in (and demonstration of) using interactive XSLT and Saxon-JS.

The screenshot in figure [fig.1] shows the edit page form of new our license tool application, rendered by Saxon-Forms.



License Tool

Relevant form fields are indicated in bold font.

Select Year: All **Redistribution (-R) only:** ☐ **Site (-S) only:** ☐ **Limit number of results** 20

Edit Form *Required fields are marked with *.*

Licensee Details

First Name*	O'Neil	**Last Name***	Delpratt
Company		**Email***	oneil@saxonica.com
Address Line 1		Address Line 2	
Town/City		County/State	
Post Code		**Country***	United Kingdom
Phone			

License Details

Start Date 31/01/2018

Expiry

Upgrade Days

Maintenance Days

Existing License

January 2018

Mon	Tue	Wed	Thu	Fri	Sat	Sun
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Invoicing Details

EU VAT No.

Order Ref

Reseller

Despatch Email

Order Details

Date Order Placed 31/01/2018

Code	Edition	Platform	Features	Quantity	Unit Price
EE-EVAL	EE	J	TQV	1	0

Select User: Default Create/Save: ✓ Send: ✓

Figure 1. The edit page of the license tool application

1.2. XForms

Forms are a common feature of interactive web applications, allowing users to enter data for submission. HTML forms can be generated in many ways: some sites serve up form pages from servers using languages such as PHP, JSP, ASP, etc. where the form submission and validation is handled on the server or via Ajax techniques. One of the greatest shortcomings of HTML forms is that the combination of presentation and the content is cumbersome and chaotic to manage. XForms was designed as a direct replacement for HTML forms to address these problems and to do much more. In XForms the presentation and content are separate, and more complicated forms can be authored using form model and controller logic.

Using XForms, a form consists of a section that describes what the form does, called the XForms *model* (contained in an `xforms:model` element, where the `xforms` prefix is used for elements in the XForms `http://www.w3.org/2002/xforms` namespace), and another section that describes how the form is to be presented. The model contains the *instance* (in an `xforms:instance` element) holding the underlying data of the form in an XML structure, *model item properties* describing declarative validation information for constraining values (in `xforms:bind` elements), and details for form data submission (in `xforms:submission` elements). The presentational part of a form contains XForms form controls (such as `input`, `select`, and `textarea` elements) which act as a direct point of user interaction, and can provide read/write access to the instance data. Typically form controls will bind to instance data nodes (as defined by an XPath expression in the `ref` attribute). Note that instance data is only presented to the user when such a binding to a form control exists; and individual form controls are only included in the user interface if the instance data node is relevant (as defined using a `relevant` attribute on an `xforms:bind` element). Actions defining event responses are specified on form controls using action elements, such as `xforms:action` and `xforms:setvalue`.

For a form-based application such as the license tool, XForms is the right choice. As described, it allows for data processing and validation in the form, and of course we want to use XML technologies and maintain our data in XML!

We decided to write a new implementation of XForms to use in our license tool, rather than using existing implementations which run in the browser, because we could see the potential for better integration of XForms into a web application which uses Saxon-JS technologies. As well as being able to use new XSLT 3.0 features, the use of Saxon-JS technologies for our new XForms implementation provides the opportunity to do more at the boundary between the XForms form and the containing application. For example in our license tool, the application logic allows parsing of structured input pasted into a text field. That's beyond the capability of XForms itself, but it can be done in XSLT, and can be integrated into what is predominantly a form-based application. So it's not just XForms; it's XForms integrated into declarative client-side applications.

1.3. XSLT 3.0 and interactive XSLT in the browser with Saxon-JS

Saxon-JS is an XSLT run-time which executes an SEF (stylesheet export file), the compiled form of an XSLT stylesheet generated using Saxon-EE. The Saxon-Forms XSLT stylesheet module is designed to be imported into the client-side XSLT stylesheet of a web application, which is exported to SEF for use with Saxon-JS. Details of how the use of XForms (via Saxon-Forms) can be integrated into the application stylesheet will be covered later. In this section, we briefly highlight the features of Saxon-JS which make it a good fit for implementing XForms:

1. XSLT 3.0 [7] (including XPath maps and dynamic evaluation)
2. Interactive XSLT - for browser event handling
3. Using global JavaScript variables and functions

In Saxon-Forms, we use a number of new XSLT 3.0 features, such as XPath maps and arrays (e.g. for actions and bindings), and dynamic evaluation of XPath with `xsl:evaluate` [8] (e.g. for XForms binding references for form controls). For further details see Section 2. Another feature of Saxon-JS which is crucial to Saxon-Forms is interactive XSLT, used to implement the dynamic interactive functionality of XForms. The interactive XSLT extensions available with Saxon-JS allow rich interactive web applications to be written directly in XSLT. Event handling templates can respond to user input; and trigger template rules to modify the content of the HTML page.

Furthermore, using the `ixsl:schedule-action` instruction with the `http-request` attribute, HTTP requests can be made, and the responses handled. See the submission example in Section 3.2 for further information on how this can be used in the integration of XForms in an application.

A few parts of the XForms implementation are done using JavaScript rather than XSLT. Using Saxon-JS, global JavaScript variables and functions are accessible within the XSLT stylesheet as functions in the `http://saxonica.com/ns/globalJS` namespace, or using the `ixsl:call()` function. Script elements can be inserted into the HTML page using interactive XSLT, providing global JavaScript functions to be used later. Global JavaScript variables are very useful as mutable objects, for example we use a JavaScript variable to hold the XForms instance as a node, this can then be easily accessed and changed to process the form interaction.

2. XForms implementation

The main work of our XForms implementation can be split into two parts, that we will refer to as *initialization* and *interaction handling*. Initialization consists of transforming the presentational part of an XForms form, to render this using HTML to correctly display the form in a browser; as well as setting up various structures which hold the details of the form (the model item properties, etc.), to be used internally by the implementation. Interaction handling involves acting on user interaction with form controls, to update the XForms instance and form display accordingly, and handling user submission which means submitting the instance to a server. In Section 2.1 we describe how we implement these two areas, using interactive XSLT 3.0. In the early stages of development, we referred to the XSLTForms implementation (which is based on XSLT 1.0 to compile XForms to (X)HTML and JavaScript) for ideas on how to get started, but using XSLT 3.0 and interactive XSLT provides many new ways of doing things and so our implementation is really written from scratch.

Following this, we briefly discuss the XForms coverage of the Saxon-Forms implementation, to give an idea of how much of the XForms specification is implemented.

2.1. Overview of how Saxon-Forms works

Initialization

XForms is designed to be integrated into other markup languages, e.g. (X)HTML. For use with Saxon-Forms, a form is supplied as an XML document, containing the XForms model and presentational part. This XForms form document is supplied via the main entry template rule of the stylesheet, named "xformsjs-main", as a template parameter. Further template parameters can be used to also supply XML instance data, and details of where the form is to appear in the HTML page (by giving the `id` of an HTML `div` element into which the rendered form will be inserted).

The result of Saxon-Forms initialization should be that the form is rendered using HTML, and inserted into the HTML page as directed. Behind the scenes, various variables have also been initialized for internal use, and these are held in the JavaScript global space, using a `script` element (with `id="xforms-cache"`) which is inserted into the HTML head. The script also includes corresponding JavaScript set/get functions for these variables. (When such functions are called from the Saxon-Forms XSLT stylesheet, e.g. using `ixsl:call()`, Saxon-JS will convert the XML items supplied as parameters into JavaScript, and convert the results back the other way, as described in [6]. Below we generally just refer to the XML types.) We cache the following variables relating to the current XForms document:

- the XForms document itself, as a node, required if we need to reset the form
- the instance in its initial state, as a node
- the instance, a node which is updated as a user interacts with the form

- *actions map*, a JSON object whose keys are unique identifiers for each action defined in the form, and the corresponding value is an XPath map which holds the details of the actions
- *relevant attributes map*, an XPath map which maps instance nodes to XPath expressions, taken from the `ref` and `relevant` attributes on `xforms:bind` elements, for example:

```
map{"Document/Options/MaintenanceDate": "../MaintenanceDateSelected='true'",
   "Document/Options/UpgradeDate": "../UpgradeDateSelected='true'", ...}
```

- *pending updates list*, an XPath map which keeps a record of updates for instance nodes which are not bound to form controls

Meanwhile, Saxon-Forms converts XForms form controls to equivalent (X)HTML form control elements (inputs, drop-down lists, textareas, etc.), populated with any bound data from the instance, and which are embellished with additional attributes containing references for use internally. For example:

```
<xforms:input incremental="true"
   ref="Document/Shipment/Order/MaintenanceDays">
   <xforms:action ev:event="xforms-value-changed">
      <xforms:setvalue ref="../../../Options/MaintenanceDate"
         value="if(xs:integer(.) &gt; 0) then
            xs:date(../../../Options/StartDate) +
            xs:dayTimeDuration(concat('P',.,'D'))
            else xs:date(../../../Options/StartDate) +
            xs:dayTimeDuration(concat('-','P',abs(xs:integer(.)),'D'))"/>
      <xforms:setvalue
        ref="../../../Options/Updated">true</xforms:setvalue>
   </xforms:action>
</xforms:input>
```

Will be converted to:

```
<input data-element="MaintenanceDays" data-constraint="number(.) ge 0"
   data-action="d26aApDhDa"
   type="text" value="30"
   data-ref="Document/Shipment/Order/MaintenanceDays"/>
```

Here, in the Saxon-Forms template rule which matches the `xforms:input` control we get the string value from the `ref` attribute, which defines the binding to an instance node, and use this XPath expression in two ways. Firstly, we call the XSLT 3.0 `xsl:evaluate` instruction to dynamically evaluate the XPath expression, to obtain the relevant data value from the instance. This will be used to populate the corresponding HTML form `input` element. Secondly, the `ref` attribute XPath expression is copied into a `data-ref` attribute added to the `input` element, to preserve the binding to the instance node. For each group of action elements in an XForms form control we add an entry to the actions map in the "xforms-cache" script element. For this actions map entry, the key will be a unique identifier, and the value is an XPath map containing all the details of the actions (e.g. from the xforms:setvalue elements, etc.) In this example, we add an entry to the actions map object with key "d26aApDhDa", and value:

```
map{"@ref": "Document/Shipment/Order/MaintenanceDays",
   "@event": "xforms-value-changed",
   "setvalue": [map{"@value": "if(xs:integer(.) > 0) then ...",
         "ref": "../../../Options/MaintenanceDate"},
      map{"value": "true",
         "ref":"../../../Options/Updated"}]}
```

Then, as in the example above, we use the data-action attribute to link the input element to its relevant entry in the actions map. The conversion, and binding preservation, of other XForms form control elements is achieved in a similar way.

Interaction handling

Interactive XSLT event handling templates are used to handle user interactions with the form, such as data input in a form field or the click of a button. The event handling templates correspond to onchange and onclick events. In figure [fig.2] we illustrate the general pipeline of the processes involved when a user interacts with the form. In this example the template rule with mode="ixsl:onchange" and match="input" is triggered when a user makes a change in an input box. Here the trigger of the template rule can only happen if the input form control has one or more actions associated with it.

Firstly, we fetch the instance XML for the form and update it with any changes made in the form controls which are not already in the instance. Secondly, we use the value in the data-action attribute on the input element to get the associated actions from the actions map. Recall that these associated actions are represented in an XPath map. So we use XPath map functions to extract the details for these actions (e.g. details for setvalue, add or delete) which are then executed. For actions which update instance nodes that are bound to form controls we first update the associated form control. Otherwise, for actions which update instance nodes which are not bound to a form control, we add the changes to the pending updates list.

Thirdly, after all actions have been executed we again update the instance XML (applying the updates in the pending updates list, and picking up changes within form controls) to maintain consistency between the data currently held in the form controls and the instance itself. The final stage is to execute the relevant properties tests for instance nodes (as defined in the relevant attributes map), to determine whether the form controls that they bind to should be included in the rendered form. The corresponding HTML form controls are hidden and revealed by setting the display style property (using the ixsl:set-property interactive XSLT extension instruction) to "none" or "inline" respectively.

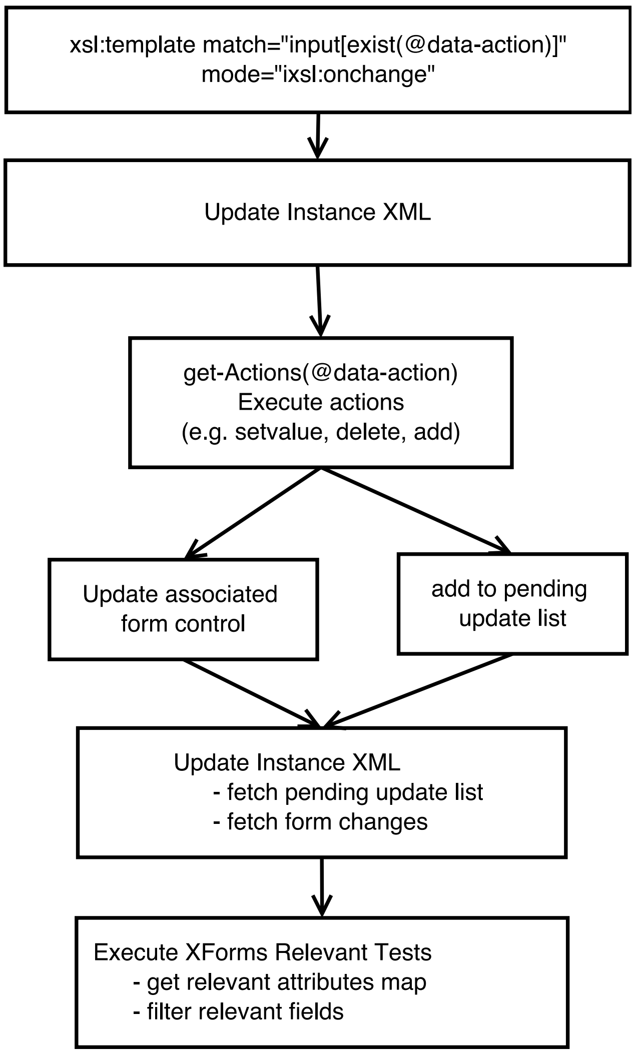

xsl:template match="input[exist(@data-action)]" mode="ixsl:onchange"
Update Instance XML
get-Actions(@data-action) Execute actions (e.g. setvalue, delete, add)
Update associated form control
add to pending update list
Update Instance XML - fetch pending update list - fetch form changes
Execute XForms Relevant Tests - get relevant attributes map - filter relevant fields

Figure 2. Action handling pipeline diagram

As well as handling changes to form data, the other key user interaction that needs to be handled is submission. However, the XForms submission element is not yet fully implemented in Saxon-Forms. One reason for this is that submission is one of the features where it is desirable, and possible, for more to be done from the application stylesheet, than could be done by a direct implementation of XForms submission. For instance, in our license tool, we use event handling templates (for onclick events on submit buttons) to override the XForms implementation for submission, in order to handle this processing and integrate handling of the server response. Further details follow in Section 3.

2.2. Coverage of the XForms Specification

Saxon-Forms is a partial implementation of the XForms 1.1 specification. The focus was on implementing the parts required to get the license tool application working. But of course it is our intention that the implementation is general enough for wider use (either used in a standalone way or as a component in an application), and has the potential to be extended for full XForms conformance. Here we summarise the main parts of the XForms specification that are implemented in Saxon-Forms, but note that in all cases (except XPath expressions) there is more which is not implemented:

- *Document structure:* Saxon-Forms currently supports just one model and one instance. In the document structure we represent the `model` element, which consists of the `instance`, `bind` and `submission` elements. This includes the type, required, constraint and relevant model item properties.
- *XPath expressions in XForms:* The specification [1] states "XForms uses XPath to address instance data nodes in binding expressions, to express constraints, and to specify calculations". Saxon-Forms is conformant to the support of XPath since Saxon-JS supports nearly all of XPath 3.1.
- *XForms Function Library:* XForms 1.1 defines a number of functions, of which Saxon-Forms currently only implements `index()` and `avg()`. These are implemented using stylesheet functions, which are then available in the static context for calls on `xsl:evaluate`. Other XForms functions could be implemented in the same way.
- *Core Form Controls:* Saxon-Forms implements the `input`, `textarea` and `select1` form control elements. Of the common support elements (child elements of the form controls), the `label` and `hint` elements are implemented. Of the container form controls (used for combining form controls), only the `repeat` element is implemented.
- *XForms Actions:* Saxon-Forms implements the `action`, `setvalue`, `insert` and `delete` elements.

3. Integrating Saxon-Forms into applications

3.1. Standard integration

Saxon-Forms includes a Saxon-JS stylesheet providing generic XSLT 3.0 code to implement the XForms specification. This can be integrated with application-specific XSLT 3.0 code. Thus, the Saxon-Forms stylesheet module can either be imported into a containing XSLT stylesheet (for the client-side of a web application), or used directly. In either case, to run in Saxon-JS, the stylesheet must first be exported to SEF using Saxon-EE. This can then be run from within an HTML page: as with all Saxon-JS applications, first Saxon-JS is loaded in a script element, and then the SEF can be executed using a JavaScript call to `SaxonJS.transform()`. An XForms document is supplied to Saxon-Forms either as a file or as a document node, along with the optional XForms instance data.

If the Saxon-Forms stylesheet is to be used directly, then the XForms document can be supplied as the source to the transform, as in the example below:

```
<script>
   window.onload = function () {
      SaxonJS.transform({
         "stylesheetLocation": "saxon-xforms.sef.xml",
         "sourceLocation": "sampleBookingForm.xml"
      })
   }
</script>
```

Alternatively, when the Saxon-Forms stylesheet module is imported into the client-side XSLT stylesheet of a web application (e.g. sample-app.xsl), this can be run as follows:

```
<script>
   window.onload = function () {
      SaxonJS.transform({
         "stylesheetLocation": "sample-app.sef.xml",
         "initialTemplate": "main"
      })
   }
</script>
```

And in this case, the XForms document can be supplied at the point that the entry template "xformsjs-main" of Saxon-Forms is called in the sample-app stylesheet:

```
<xsl:template name="call-saxon-forms">
   <xsl:call-template name="xformsjs-main" >
      <xsl:with-param name="xforms-doc" select="doc($bookingForm)"/>
      <xsl:with-param name="xFormsId" select="'xForm'"/>
   </xsl:call-template>
</xsl:template>
```

Here the `xFormsId` parameter gives the `id` of a `div` element in the HTML page where the form is to be inserted; the default is "xForm".

3.2. Integration with application logic

Saxon-Forms is more than just another XForms implementation for the browser, because it allows for form enrichment from application logic in the application stylesheet in which it is integrated. In this section we will present some examples of this:

1. Parsing structured text from a form input textarea, to XML.
2. Overriding submission.
3. Using user defined functions in XPath expressions in the XForm.

Example 1. Parsing input from form textareas

This has proved very useful in our license tool. License orders are often received by email using structured text of a standard form (e.g. for purchases from the online shop, and for evaluation license requests). Because the text is structured, it can be processed using XSLT to extract the data and convert it into XML format. So this parsing can be done in the application stylesheet.

So, a user copies the structured text from an email and inputs it into the textarea of a form in the tool. When the "Parse" button is clicked, this is handled by event handling templates in the application stylesheet which capture the text string and process it to produce some XML output. This XML is then supplied as the instance for another XForms form (in fact, the edit page form, as shown in [fig.1]).

Example 2. Submission

The XForms implementation for submission can be overridden from the application stylesheet, to allow further logic to be added to specify the exact form of the submitted data, and the way a response is handled. For example in the license tool stylesheet, we have event handling templates for onclick events on submit buttons to handle this processing. The updated instance is obtained from the global JavaScript variable (using the procedure in the Saxon-Forms submission implementation), and this is submitted for server side processing using the interactive XSLT mechanism for asynchronous HTTP messages, i.e. using the `ixsl:schedule-action` instruction with the `http-request` attribute. The value of the `http-request` attribute is an XPath map which defines the HTTP request to be made (e.g. specifying method, URI destination, body and media-type). When it returns, the HTTP response is processed by the template specified within the `ixsl:schedule-action` instruction (it has one `xsl:call-template` child); the HTTP response is also represented as an XPath map, and this is provided as the context item to the named template. For instance, this allows feedback from the response to then be returned to the user within the HTML page.

Example 3. User defined functions

Stylesheet functions defined in the application stylesheet can be used in XPath expressions in the XForms document. The only requirement is that the saxon-xforms.xsl stylesheet must include a namespace declaration binding the prefix used in the form to the namespace of the stylesheet function.

For example, the following stylesheet function is defined in our license tool application stylesheet, to obtain product price data from another XML document:

```
<xsl:function name="f:productCodeToPrice" as="xs:integer">
   <xsl:param name="productCode" as="xs:string"/>
   <xsl:variable name="products" select="doc($productsDoc)//Product"/>
```

```
    <xsl:value-of select="xs:integer($products[@code = $productCode]/▶
@price)"/>
</xsl:function>
```

This function can then be used in the XPath expressions in the `value` attribute of a `xforms:setvalue` instruction in the XForm document, to calculate the order part value from the price and quantity (where parts of an order are grouped by product code).

4. Conclusion

In this paper we have presented a new XForms implementation, Saxon-Forms, which makes use of interactive XSLT 3.0 to realize the initialization and processing model of XForms. This project had three goals:

- Firstly, our aim was to explore how XForms and client-side XSLT could coexist to build applications with rich client-side functionality as well as access to server-side functions.
- Secondly, to develop the beginnings of a new XForms implementation taking advantage of the Saxon-JS technology, and able to integrate with Saxon-JS applications.
- Thirdly, to use this technology platform to re-engineer the in-house Saxon license tool application.

  Our achievements so far against these goals are:

1. We have demonstrated that a forms-based application can be usefully augmented with additional functionality implemented in XSLT 3.0, for example parsing and validation of complex input fields, and access to reference datasets.
2. We have shown that many of the technical features of the Saxon-JS technology, such as the ability to handle interactive user input using template rules, the ability to issue asynchronous HTTP requests and process the results, and the ability to dynamically evaluate XPath expressions, can be exploited as underpinnings to a client-side XForms implementation.
3. We have rewritten the Saxon license tool application with many new features, with 90% of the code now being in either client-side or server-side XSLT, reducing the Java to a small number of extension functions handling cryptographic signing of licenses.

Further work taking this technology forwards to a fully compliant XForms implementation will depend on user feedback.

5. Acknowledgements

Many thanks to Michael Kay and Alain Couthures for helpful comments for improving this paper, and Saxon-Forms itself.

Bibliography

[1] *XForms 1.1 Specification. W3C Recommendation.* 20 October 2009. John Boyer. W3C. `https://www.w3.org/TR/xforms11`

[2] *XSLTForms*. Alain Couthures. `http://www.agencexml.com/xsltforms`

[3] *Saxon-JS: XSLT 3.0 in the Browser*. *Balisage: The Markup Conference 2016.* Debbie Lockett and Michael Kay. `http://www.balisage.net/Proceedings/vol17/html/Lockett01/BalisageVol17-Lockett01.html`

[4] *Distributing XSLT Processing between Client and Server*. O'Neil Delpratt and Debbie Lockett. XML London. June, 2017. London, UK. `http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf#page=8`

[5] *Interactive XSLT in the browser*. *Balisage: The Markup Conference 2013.* O'Neil Delpratt and Michael Kay. `https://www.balisage.net/Proceedings/vol10/html/Delpratt01/BalisageVol10-Delpratt01.html`

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

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

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

Life, the Universe, and CSS Tests

Tony Graham
Antenna House, Inc.
`<tony@antennahouse.com>`

Abstract

The W3C CSS Working Group maintains a CSS test suite already composed of more than 17,000 tests and growing constantly. Tracking the results of running such a large number of tests on a PDF formatter is more than anyone could or should want to do by hand. The system needs to track when a test's result changes so that the changes can be verified and the test's status updated. Finding differences is not the same as checking correctness. An in-house system for running the tests and tracking their results has been implemented as an eXist-db app. Is it a masterpiece of agile development, or an example of creeping featurism?

1. Introduction

"That's right," shouted Vroomfondel, "we demand rigidly defined areas of doubt and uncertainty!"

—The Hitchhiker's Guide to the Galaxy, Douglas Adams

This paper describes an internal project to develop a system for:

- Running the CSS WG test suite of more than 17,000 tests on AH Formatter;
- Producing PDF output; and
- Recording the status of the test results.

Just as importantly, the system needs to track whenever a test's result changes so that the changes can be verified and the test's status updated.

Finding differences is **not** the same as checking correctness. The first time that you look at a test's result, you can (hopefully) tell if it is right or wrong. If the result changes, either because of changes in the formatter or because of changes in the test itself, you need to look at it again since:

- The result could still be right;
- The result could still be wrong;
- The result could change from right to wrong; or, more preferably,
- The result could change from wrong to right.

2. Origins

In the beginning the Universe was created.
This has made a lot of people very angry and been widely regarded as a bad move.
—*The Restaurant at the End of the Universe*, Douglas Adams

The current system has multiple origins or predecessors:

- Antenna House Regression Testing System (AHRTS) – Software for comparing two PDFs or images – or two whole directories containing PDFs or images – and producing an overview report and plus individual reports for each pair of files with differences.
- Customized AHRTS reports – Modifications and additions made to the default stylesheets for generating PDF of AHRTS reports from XML source.
- CSS Working Group test results – The CSS WG have their own format for recording the status of a browser's results for the CSS WG tests.
- SVG and MathML test results – Previous company-internal tests of formatting of both SVG and MathML had their results recorded in a manually-maintained HTML file.
- XSL 1.0 Candidate Recommendation test results – The XML format for recording a test's result allowed recording both an indication of the result's state plus a comment about the test result.
- eXist-db demo application – The current eXist-db app started out by copying and modifying the demo app provided with eXist-db 3.0.0.

2.1. Antenna House Regression Testing System (AHRTS)

Out of the box, AHRTS [2] makes a pixel-by-pixel, visual comparison of the differences between the PDF (or image) files in a 'base' directory against the same-named files in a 'new' directory. It produces an overview PDF report listing the state of each test file plus an individual PDF report for each test file with differences. Each individual PDF report contains some data about the 'base' and 'new' test files plus, for each page with differences, a page showing: the 'base' page; a composite of the 'base' and 'new' pages with the differences highlighted; and the 'new' page.

AHRTS can be run from a GUI or through the command line. As Figure 1 shows, its additional inputs are an 'ahrts.properties' file for controlling the operation plus the XSLT stylesheets for the overview and individual reports.

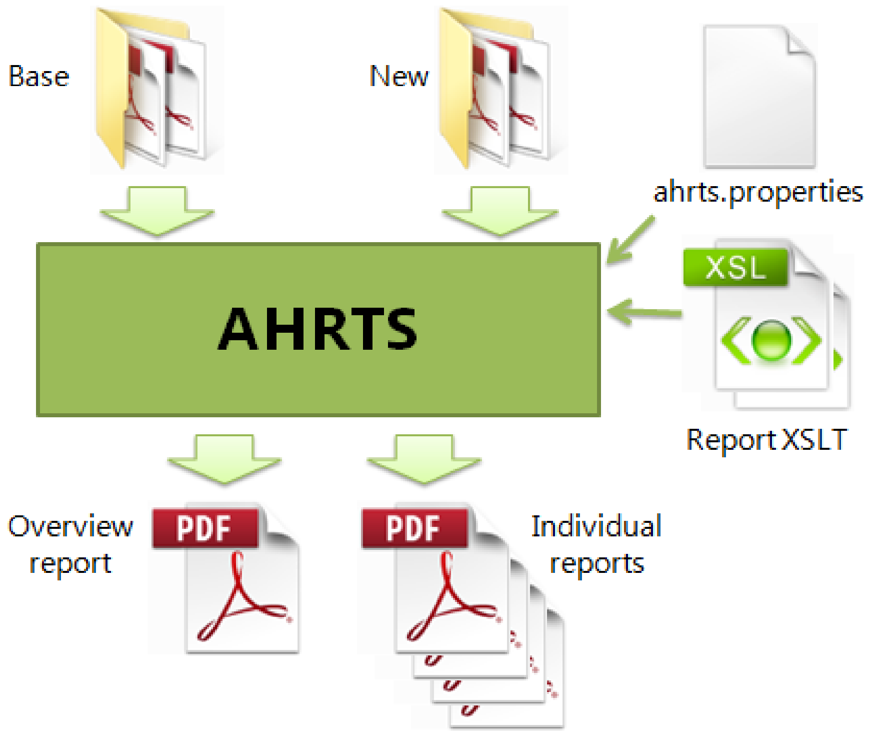
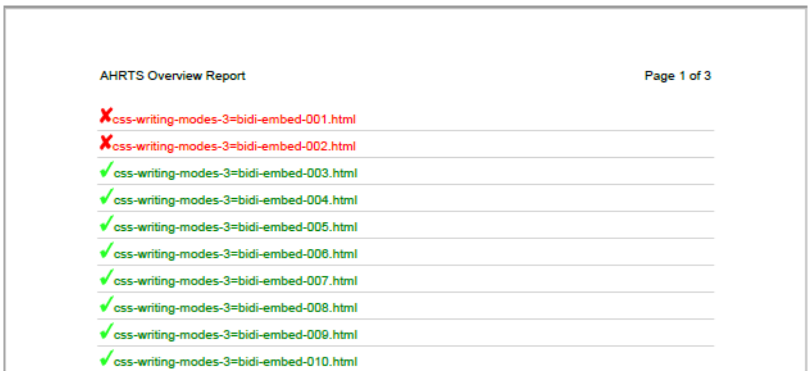


Figure 1. AHRTS block diagram

Figure 2 shows part of an AHRTS overview report with its indications of which files showed differences.


AHRTS Overview Report Page 1 of 3

✗ css-writing-modes-3=bidi-embed-001.html
✗ css-writing-modes-3=bidi-embed-002.html
✓ css-writing-modes-3=bidi-embed-003.html
✓ css-writing-modes-3=bidi-embed-004.html
✓ css-writing-modes-3=bidi-embed-005.html
✓ css-writing-modes-3=bidi-embed-006.html
✓ css-writing-modes-3=bidi-embed-007.html
✓ css-writing-modes-3=bidi-embed-008.html
✓ css-writing-modes-3=bidi-embed-009.html
✓ css-writing-modes-3=bidi-embed-010.html

Figure 2. AHRTS overview report (detail)

Figure 3 shows one page from an individual report. The same page from the 'base' and 'new' files are shown on the left and right panels, respectively. The center panel is an overlay of the 'base' and 'new' pages with their differences highlighted.

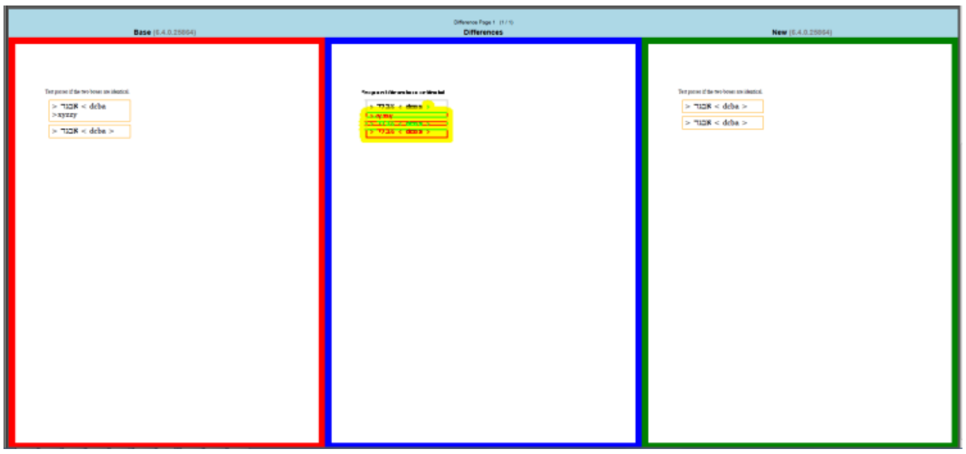


Figure 3. AHRTS individual report differences page

Figure 4 illustrates the overlaying of the 'base' and 'new' results and the highlighting of their differences.

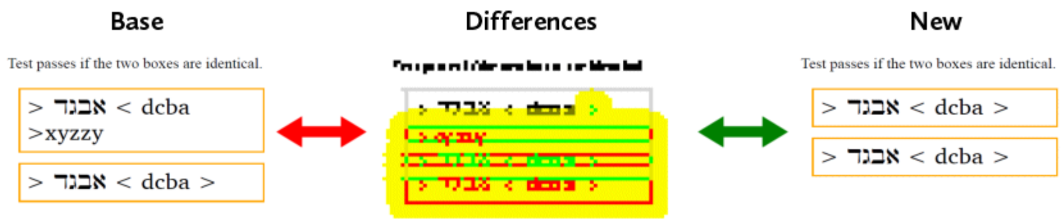

Base
Test passes if the two boxes are identical.
> אבגד < dcba
>xyzzy
> אבגד < dcba >
Differences
New
Test passes if the two boxes are identical.
> אבגד < dcba >
> אבגד < dcba >

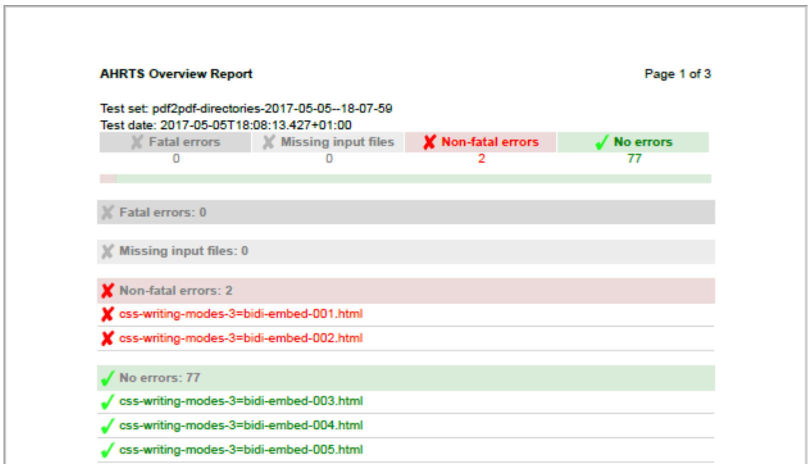
Figure 4. AHRTS difference reporting

2.2. Customized AHRTS Reports

When I started working with AHRTS, it was to check the effect of my changes to XSL-FO processing. I didn't want to look through pages of results to spot the ones with differences, so I used AHRTS and I used the Jenkins Continuous Integration Server to automate the running of both the formatter and AHRTS.

AHRTS generates its listing of differences as an XML file, and its PDF reports are produced by using XSLT to generate XSL-FO that is formatted using a built-in version of AH Formatter. Since the presentation aspects come from the XSLT, I also made an alternative XSLT stylesheet that groups tests by their results. From there, it wasn't much more effort to add counts of each result type.

Figure 5 shows a portion of the overview report produced using the 'alternative' stylesheet that is included with AHRTS 1.4.



AHRTS Overview Report — Page 1 of 3

Test set: pdf2pdf-directories-2017-05-05--18-07-59
Test date: 2017-05-05T18:08:13.427+01:00

✗ Fatal errors	✗ Missing input files	✗ Non-fatal errors	✓ No errors
0	0	2	77

✗ Fatal errors: 0

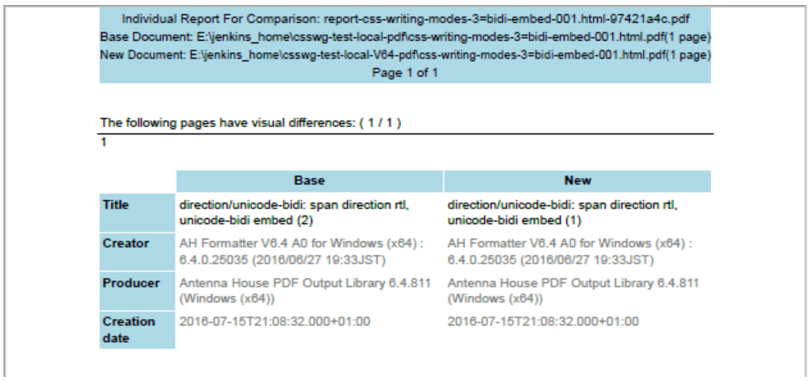
✗ Missing input files: 0

✗ Non-fatal errors: 2
- ✗ css-writing-modes-3=bidi-embed-001.html
- ✗ css-writing-modes-3=bidi-embed-002.html

✓ No errors: 77
- ✓ css-writing-modes-3=bidi-embed-003.html
- ✓ css-writing-modes-3=bidi-embed-004.html
- ✓ css-writing-modes-3=bidi-embed-005.html

Figure 5. Alternate AHRTS overview report (detail)

With AHRTS 1.4, it's now possible to include metadata from the 'base' and 'new' PDFs in the individual report, as Figure 5 shows.



Individual Report For Comparison: report-css-writing-modes-3=bidi-embed-001.html-97421a4c.pdf
Base Document: E:\jenkins_home\csswg-test-local-pdf\css-writing-modes-3=bidi-embed-001.html.pdf(1 page)
New Document: E:\jenkins_home\csswg-test-local-V64-pdf\css-writing-modes-3=bidi-embed-001.html.pdf(1 page)
Page 1 of 1

The following pages have visual differences: (1 / 1)
1

	Base	New
Title	direction/unicode-bidi: span direction rtl, unicode-bidi embed (2)	direction/unicode-bidi: span direction rtl, unicode-bidi embed (1)
Creator	AH Formatter V6.4 A0 for Windows (x64) : 6.4.0.25035 (2016/06/27 19:33JST)	AH Formatter V6.4 A0 for Windows (x64) : 6.4.0.25035 (2016/06/27 19:33JST)
Producer	Antenna House PDF Output Library 6.4.811 (Windows (x64))	Antenna House PDF Output Library 6.4.811 (Windows (x64))
Creation date	2016-07-15T21:08:32.000+01:00	2016-07-15T21:08:32.000+01:00

Figure 6. Alternate AHRTS individual report (detail)

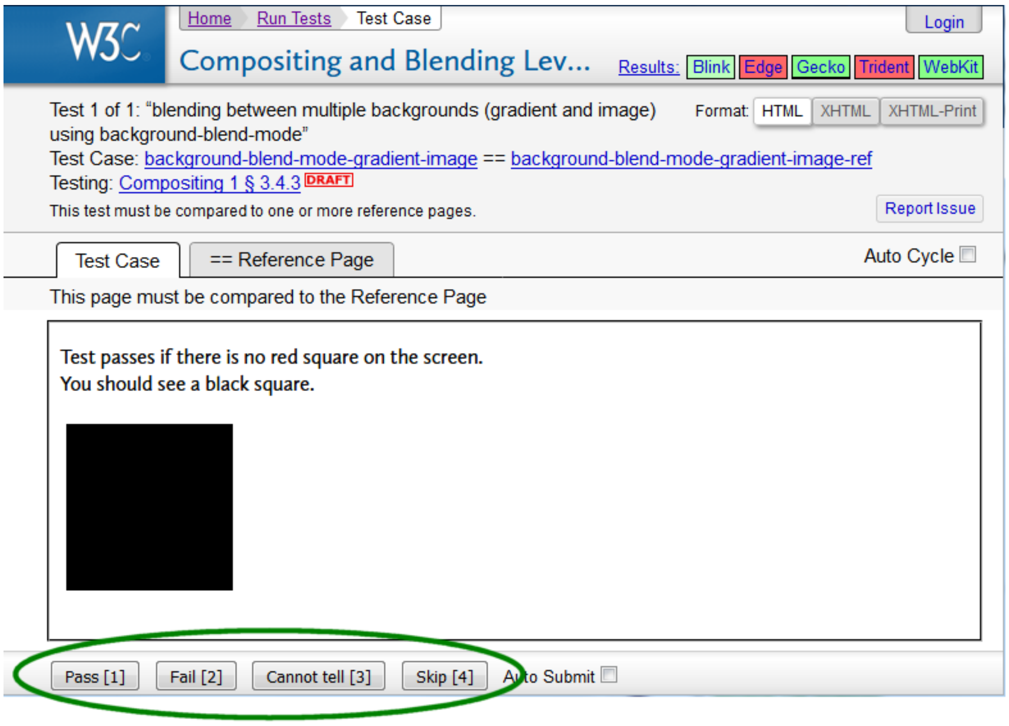
The XSLT stylesheet can be set in the AHRTS GUI's 'Settings' tab, in the AHRTS properties file, or on the AHRTS command line.

2.3. CSS Test Suite Results

When it was time to check the CSS features defined in some of the newer CSS modules that were stable enough to be implemented, we looked again at how the CSS WG reports its results.

The CSS WG has a comprehensive test suite of about 17,000 test files (and growing) and has at least two test harnesses for looking at tests in a browser and reporting results. The test harnesses obviously aren't unusable when producing PDF.

Figure 7 is a screenshot of a page from the main CSS test harness in action. The buttons for selecting the status of the test result are highlighted.



Test 1 of 1: "blending between multiple backgrounds (gradient and image) using background-blend-mode"
Test Case: background-blend-mode-gradient-image == background-blend-mode-gradient-image-ref
Testing: Compositing 1 § 3.4.3 DRAFT
This test must be compared to one or more reference pages.

This page must be compared to the Reference Page

Test passes if there is no red square on the screen.
You should see a black square.

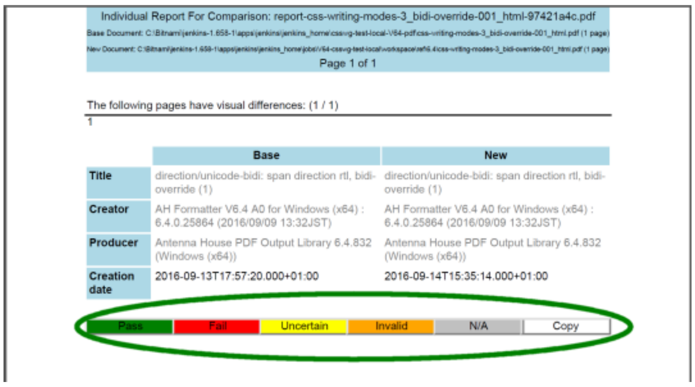
Figure 7. CSS test harness

The CSS WG reports test results as one of five categories [1]:

pass	Test passes
fail	Test fails
na	Test does not apply
invalid	Test is invalid
?	Can't tell or don't know

Since I was already using Jenkins to run both AH Formatter and AHRTS, rather than adding or writing yet another application, I wanted a simple way to use Jenkins to collect CSS test results. I made a version of the XSLT for individual reports that added a set of links, one for each CSS test result state plus a sixth that just copies the test's PDF file. Each link triggers the same 'testresult' Jenkins job but provides different parameters, most noticeably the parameter indicating the test result state.

Figure 8 shows a portion of an individual AHRTS report with the links for the test results highlighted.



Individual Report For Comparison: report-css-writing-modes-3_bidi-override-001_html-97421a4c.pdf

Base Document: C:\Bitnami\jenkins-1.658-1\apps\jenkins\jenkins_home\csswg-test-local-V64-pdf\css-writing-modes-3_bidi-override-001_html.pdf (1 page)

New Document: C:\Bitnami\jenkins-1.658-1\apps\jenkins\jenkins_home\jobs\V64-csswg-test-local\workspace\ref6.4\css-writing-modes-3_bidi-override-001_html.pdf (1 page)

Page 1 of 1

The following pages have visual differences: (1 / 1)

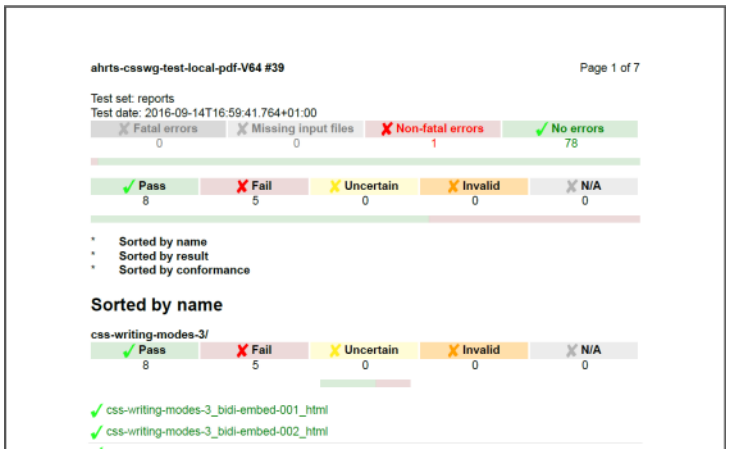
1

	Base	New
Title	direction/unicode-bidi: span direction rtl, bidi-override (1)	direction/unicode-bidi: span direction rtl, bidi-override (1)
Creator	AH Formatter V6.4 A0 for Windows (x64) : 6.4.0.25864 (2016/09/09 13:32JST)	AH Formatter V6.4 A0 for Windows (x64) : 6.4.0.25864 (2016/09/09 13:32JST)
Producer	Antenna House PDF Output Library 6.4.832 (Windows (x64))	Antenna House PDF Output Library 6.4.832 (Windows (x64))
Creation date	2016-09-13T17:57:20.000+01:00	2016-09-14T15:35:14.000+01:00

Figure 8. AHRTS individual report with CSS result links

The ‘testresult’ Jenkins job simply runs an Ant task that appends the supplied parameter values as a new line in a log file and also copies the test’s PDF to the test job’s ‘base’ directory[1]. The next run of the AHRTS job uses the log file information to display the test’s state alongside the indication of whether the current result is the same as the base. And, since I already had similar plumbing for counting tests with differences, the summary report also showed counts of the reports with each state.

Figure 9 shows a portion of an overview report showing both counts of the differences found by AHRTS and counts for each test result state.



ahrts-csswg-test-local-pdf-V64 #39 Page 1 of 7

Test set: reports
Test date: 2016-09-14T16:59:41.764+01:00

✗ Fatal errors	✗ Missing input files	✗ Non-fatal errors	✓ No errors
0	0	1	78

✓ Pass	✗ Fail	✗ Uncertain	✗ Invalid	✗ N/A
8	5	0	0	0

- Sorted by name
- Sorted by result
- Sorted by conformance

Sorted by name

css-writing-modes-3/

✓ Pass	✗ Fail	✗ Uncertain	✗ Invalid	✗ N/A
8	5	0	0	0

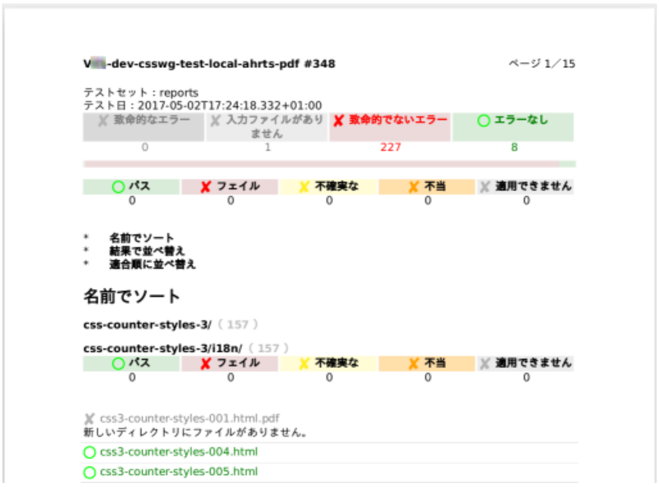
✓ css-writing-modes-3_bidi-embed-001_html
✓ css-writing-modes-3_bidi-embed-002_html

Figure 9. AHRTS overview report with CSS status (detail)

2.3.1. Localization

Along the way, I also implemented some localization functions in XSLT to make it easy to generate AHRTS reports in Japanese for use by colleagues in Japan. Figure 10 shows a similar portion of an overview report localized for Japanese.

[1]Ant works the same on both Linux and Windows, so using Ant avoids having to write both a Linux-specific and a Windows-specific version of the same script.



V[redacted]-dev-csswg-test-local-ahrts-pdf #348 ページ 1／15

テストセット：reports
テスト日：2017-05-02T17:24:18.332+01:00

✗ 致命的なエラー	✗ 入力ファイルがありません	✗ 致命的でないエラー	○ エラーなし
0	1	227	8

○ パス	✗ フェイル	✗ 不確実な	✗ 不当	✗ 適用できません
0	0	0	0	0

* 名前でソート
* 結果で並べ替え
* 適合順に並べ替え

名前でソート

css-counter-styles-3/ (157)

css-counter-styles-3/i18n/ (157)

○ パス	✗ フェイル	✗ 不確実な	✗ 不当	✗ 適用できません
0	0	0	0	0

✗ css3-counter-styles-001.html.pdf
新しいディレクトリにファイルがありません。

○ css3-counter-styles-004.html

○ css3-counter-styles-005.html

Figure 10. Japanese AHRTS overview report with CSS status (detail)

AHRTS is written in Scala, and it uses Java property files in either textual or XML format for run-time lookup of localized strings. AHRTS's XML property localization files are installed with AHRTS, so it was easy to write XSLT that would work with them.

The localization XSLT functions support lookup of fixed strings:

```
<fo:block>
  <xsl:value-of select="axf:l10n('Test set: ')" />
  <xsl:value-of select="overview/@test-set" />
</fo:block>
```

which can be subverted to look up data in addition to the text that appears in the formatted output:

```
<fo:simple-page-master
    master-name="report-page"
    page-height="{axf:l10n('-page-height')}"
    page-width="{axf:l10n('-page-width')}">
```

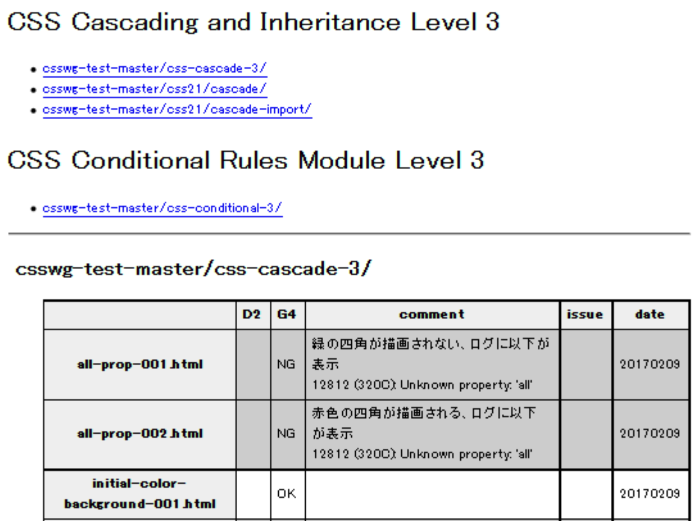
It also supports positional parameters for when the sentence structure differs between languages:

```
<xsl:copy-of
    select="axf:l10n('Page %1 of %2',
                     ($fo-page-number,
                      $fo-page-number-citation-last))" />
```

2.4. HTML Report

However, and there's always a 'however', I was now told that my colleagues in Japan wanted an HTML report. Producing an HTML version of the current overview report was a straightforward reworking of parts of the existing XSL-FO stylesheet. The HTML stylesheet reused some of the existing XSLT modules that are more concerned with logic than with presentation. I was then told that they wanted a report in a format similar to that which had been used previously when testing SVG and MathML support, and they provided a copy of their current CSS test results.

Figure 11 shows a portion of the HTML report being produced by staff in Japan.


CSS Cascading and Inheritance Level 3

- csswg-test-master/css-cascade-3/
- csswg-test-master/css21/cascade/
- csswg-test-master/css21/cascade-import/

CSS Conditional Rules Module Level 3

- csswg-test-master/css-conditional-3/

csswg-test-master/css-cascade-3/

	D2	G4	comment	issue	date
all-prop-001.html		NG	緑の四角が描画されない、ログに以下が表示 12812 (320C): Unknown property: 'all'		20170209
all-prop-002.html		NG	赤色の四角が描画される、ログに以下が表示 12812 (320C): Unknown property: 'all'		20170209
initial-color-background-001.html		OK			20170209

Figure 11. Manually produced HTML report (detail)

Their report just recorded the state of the test as ‘OK’ or ‘NG’ (No Good). The real ‘however’, however, was the possible additional comment about the test, issue number, and status of the issue’s resolution.

Adding the extra fields to the PDF for an individual report was straightforward: instead of using just links, the test results were captured using an Acrobat form.

So far, so good, but this had four problems:

- I couldn’t find a PDF reader for Linux that would submit the form, so had to use Acrobat Reader on Windows.
- Acrobat would store every HTTP response from Jenkins in a different temporary file, and, for every response, Acrobat would pop-up a dialog box asking permission to open the file. Acrobat Reader could be made to trust remote sites, but apparently it can’t be set to trust local files.
- Acrobat also views filling in the form as a change to the file, so it wasn’t possible to close the file without Acrobat prompting to save the ‘changed’ file. I’ve since been advised of a way to stop this, but by then I had already moved on to using eXist-db.

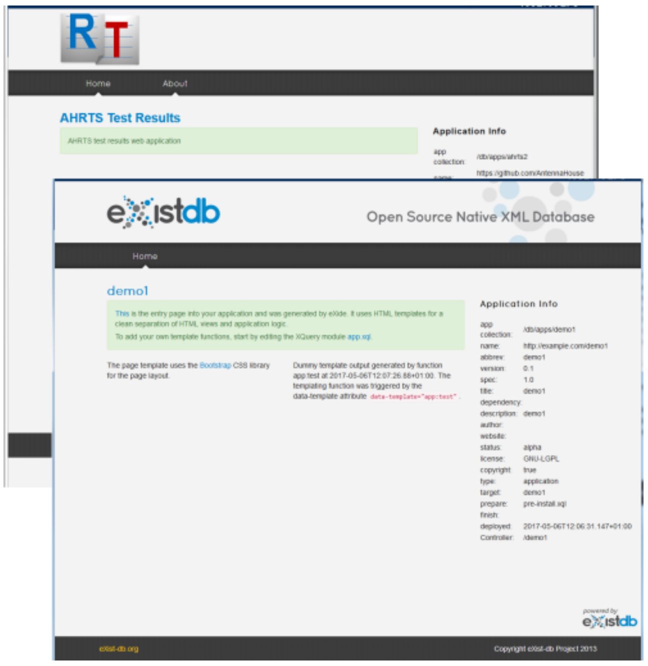
- Acrobat could submit the form to Jenkins, and Jenkins could pass ASCII data to Ant okay, but Japanese text in the comment field was garbled in a way that I couldn't decipher.

The encoding issue was the killer issue. It completely ruled out Jenkins for collecting test results, even though Jenkins was still wanted for compiling the code and running the tests. Collecting test results in a text file had always been a temporary solution. The intention had always been to move to using an XML database once the data was complex enough to justify doing so. The data still wasn't particularly complex, but the need to preserve the Japanese text made a good reason to change.

2.5. eXist-db

So the project moved to using eXist-db [5]. I chose eXist-db partly because I was more familiar with it than with BaseX, but also because I'd had more contact with the eXist-db developers so I knew who to ask if I had problems. I did have problems, but eXist-db has an active and helpful mailing list as well as developers who respond quickly to GitHub issues.

My approach to developing the eXist-db code was initially to copy and modify one of eXist-db's demo apps. This worked, but the eXist-db documentation has evolved over time, and older documentation advises separate XQuery modules in the 'modules' collection, whereas newer documentation favors (almost) all XQuery code for the app in a single 'modules/app.xql' file.


Home
About
Open Source Native XML Database
Home
powered by eXistdb
exist-db.org
Copyright exist-db Project 2013

AHRTS Test Results

AHRTS test results web application

Application Info

app collection: /db/apps/ahrts2
name: https://github.com/AntennaHouse

demo1

This is the entry page into your application and was generated by eXide. It uses HTML templates for a clean separation of HTML views and application logic.
To add your own template functions, start by editing the XQuery module app.xql.

The page template uses the Bootstrap CSS library for the page layout.

Dummy template output generated by function app:test at 2017-05-06T12:07:26.88+01:00. The templating function was triggered by the data-template attribute `data-template="app:test"`.

Application Info

app collection:	/db/apps/demo1
name:	http://example.com/demo1
abbrev:	demo1
version:	0.1
spec:	1.0
title:	demo1
dependency:	
description:	demo1
author:	
website:	
status:	alpha
license:	GNU-LGPL
copyright:	true
type:	application
target:	demo1
prepare:	pre-install.xql
finish:	
deployed:	2017-05-06T12:06:31.147+01:00
Controller:	/demo1

Figure 12. App 'About' page and default page for a new application

The initial attempt to use eXist-db was by inserting a link to the eXist-db 'app' in the PDF of an individual test result. I had also tried making eXist-db return a 206 HTTP response code and no response body (to avoid one of the problems with Acrobat) but I couldn't get that to work.

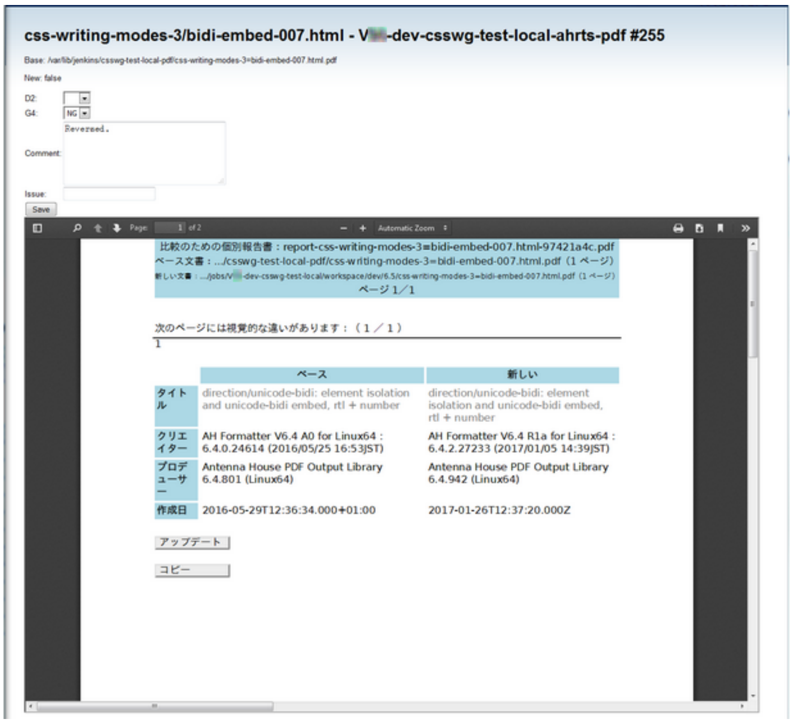
3. eXist-db Application

Share and enjoy!

—*Sirius Cybernetics Corporation motto*

An XML database could solve the problem of how to store the data about the tests, but that didn't solve the rest of the problems with the form in the PDF file. The usability breakthrough came when, instead of putting the form in the PDF, I put the PDF inside the form and created an eXist-db 'app' for reviewing test results in a web browser.

Figure 13 shows the first version of an individual report served from eXist-db, and Figure 14 shows a more recent version of an individual report.


css-writing-modes-3/bidi-embed-007.html - V[redacted]-dev-csswg-test-local-ahrts-pdf #255

Base: /var/lib/jenkins/csswg-test-local-pdf/css-writing-modes-3=bidi-embed-007.html.pdf

New: false

D2:

G4: NG

Comment: Reversed.

Issue:

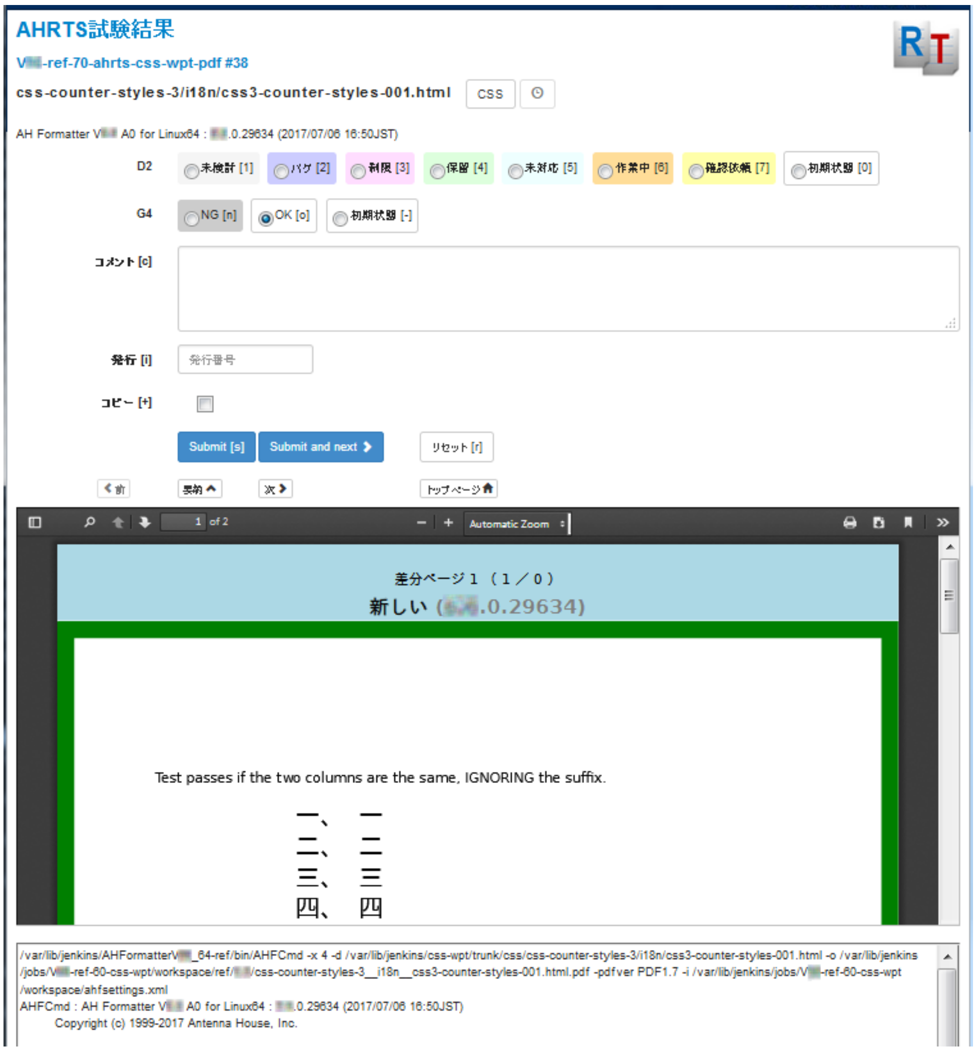
比較のための個別報告書：report-css-writing-modes-3=bidi-embed-007.html-97421a4c.pdf
ベース文書：.../csswg-test-local-pdf/css-writing-modes-3=bidi-embed-007.html.pdf（1 ページ）
新しい文書：.../jobs/V[redacted]-dev-csswg-test-local/workspace/dev/6.5/css-writing-modes-3=bidi-embed-007.html.pdf（1 ページ）
ページ 1／1

次のページには視覚的な違いがあります：（1／1）

1

	ベース	新しい
タイトル	direction/unicode-bidi: element isolation and unicode-bidi embed, rtl + number	direction/unicode-bidi: element isolation and unicode-bidi embed, rtl + number
クリエイター	AH Formatter V6.4 A0 for Linux64 : 6.4.0.24614 (2016/05/25 16:53JST)	AH Formatter V6.4 R1a for Linux64 : 6.4.2.27233 (2017/01/05 14:39JST)
プロデューサー	Antenna House PDF Output Library 6.4.801 (Linux64)	Antenna House PDF Output Library 6.4.942 (Linux64)
作成日	2016-05-29T12:36:34.000+01:00	2017-01-26T12:37:20.000Z

Figure 13. Initial individual result page (reduced)



AHRTS試験結果

V[redacted]-ref-70-ahrts-css-wpt-pdf #38

css-counter-styles-3/i18n/css3-counter-styles-001.html CSS

AH Formatter V[redacted] A0 for Linux64 : [redacted].0.29634 (2017/07/06 16:50JST)

D2: 未検討 [1] バグ [2] 制限 [3] 保留 [4] 未対応 [5] 作業中 [6] 確認依頼 [7] 初期状態 [0]

G4: NG [n] [✓ OK [o]] 初期状態 [-]

コメント [c]

発行 [i] 発行番号

コピー [+]

Submit [s] Submit and next リセット [r]

前 要約 次 トップページ

1 of 2 Automatic Zoom

差分ページ１（１／０）
新しい ([redacted].0.29634)

Test passes if the two columns are the same, IGNORING the suffix.

一、 一
二、 二
三、 三
四、 四

```
/var/lib/jenkins/AHFormatterV[redacted]_64-ref/bin/AHFCmd -x 4 -d /var/lib/jenkins/css-wpt/trunk/css/css-counter-styles-3/i18n/css3-counter-styles-001.html -o /var/lib/jenkins
/jobs/V[redacted]-ref-60-css-wpt/workspace/ref/[redacted]/css-counter-styles-3__i18n__css3-counter-styles-001.html.pdf -pdfver PDF1.7 -i /var/lib/jenkins/jobs/V[redacted]-ref-60-css-wpt
/workspace/ahfsettings.xml
AHFCmd : AH Formatter V[redacted] A0 for Linux64 : [redacted].0.29634 (2017/07/06 16:50JST)
        Copyright (c) 1999-2017 Antenna House, Inc.
```

Figure 14. Later individual result page (reduced)

3.1. Loading

A sequence of Jenkins jobs runs AH Formatter on the CSS tests then runs AHRTS to compare the latest result with a set of 'base' PDF files. The Jenkins job that runs AHRTS also uploads the AHRTS-generated XML files to eXist-db.

The XML for the overview report begins:

```
<overview date="2017-05-05T20:36:42.638+01:00"
    overview-report-title="Vxx-ref-70-ahrts-csswg-test-pdf #252"
    test-set="reports">
  <compare name="WOFF2-UserAgent=Tests=xhtml1=available-001.xht"
      module="WOFF2-UserAgent"
      missing-input-file="false" error-detected="false"
      individual-report-unicode-safe-filename-pdf=
         "000000001_97421a4c.pdf"
      fatal-error="false" difference-detected="true"
```

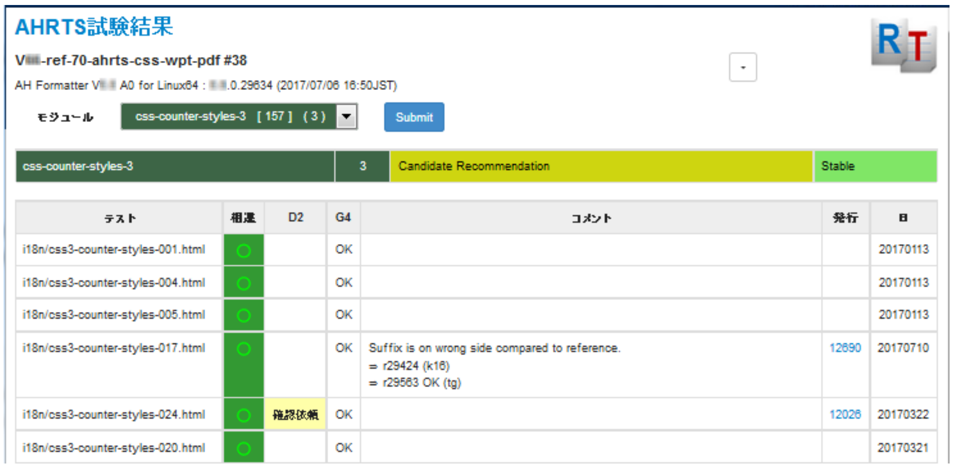
```
        individual-report-pdf=
  "report-WOFF2-UserAgent=Tests=xhtml1=available-001.xht-97421a4c.pdf">
      <warn name="pdf-annotation">どちらの PDF にも注釈タグが含まれています。
      タグは、レポートに埋め込まれた PDF には表示されませんが、比較に含まれます。
  </warn>
    </compare>
```

This XML is augmented before being uploaded to eXist-db to add the log from AH Formatter and to pre-compute some values.

The individual PDF files from AHRTS are also uploaded into the database. Storing PDF is arguably not a good use for an XML database, but it is much simpler than storing the PDFs elsewhere on the application server and then using URL rewriting to access them. My colleagues in Japan operate their own eXist-db instance, which was set up for them by their IT support staff with whom I have no contact. Since no-one in the Japan office had used eXist-db before, keeping the database installation as simple as possible is, for the moment, more important than shaving a few milliseconds off the time to serve a two-page PDF file.

3.2. Summary view

It is straightforward to use XQuery to generate a summary page from the overview XML. Early versions of the application generated a single HTML page with results for every test. Following a request by my colleagues in Japan, more recent versions present one module at a time, as shown in Figure 15:



AHRTS試験結果

V[redacted]-ref-70-ahrts-css-wpt-pdf #38

AH Formatter V[redacted] A0 for Linux64 : [redacted].0.29634 (2017/07/06 16:50JST)

モジュール css-counter-styles-3 [157] (3) Submit

css-counter-styles-3	3	Candidate Recommendation	Stable

テスト	相違	D2	G4	コメント	発行	日
i18n/css3-counter-styles-001.html	○		OK			20170113
i18n/css3-counter-styles-004.html	○		OK			20170113
i18n/css3-counter-styles-005.html	○		OK			20170113
i18n/css3-counter-styles-017.html	○		OK	Suffix is on wrong side compared to reference. = r29424 (k16) = r29563 OK (tg)	12690	20170710
i18n/css3-counter-styles-024.html	○	確認依頼	OK		12026	20170322
i18n/css3-counter-styles-020.html	○		OK			20170321

Figure 15. Summary page

In theory, every top-level directory in the CSS tests corresponds to a same-named CSS Recommendation or Working Draft. In practice, some of the directory names differ from the short name of the module they test. Also, CSS 2.1 has nearly 10,000 tests, so the subdirectories of the `CSS2` directory – `CSS2/colors`, `CSS2/fonts`, and so on – are treated as separate modules just to keep things more manageable[2].

3.3. Individual test results

A sample `testresults.xml` file containing the information recorded about the results of a single test is shown below:

```
<testresult date="20170630"
    ahf-version="AH Formatter Vx.x A0 for Linux64 : x.x.0.29482
(2017/06/27 09:25JST)">
    <d2/>
    <g4>OK</g4>
    <comment>r28137 : NG (Letters of the "Don't Panic" aren't friendly
enough.)  r29557 : OK</comment>
    <issue>12345</issue>
</testresult>
```

The format of the XML was determined, firstly, by the information that was already being recorded by my colleagues in Japan (see Section 2.4) and, secondly, by needing a simple, 'XForms-able' form for the XML. eXist-db ships with two XForms implementations [6] – XSLTForms, which works client-side, and betterFORM, which works server-side. Indeed, the template XForms instance existed before the first results could be added to the database.

Once the XML format was decided, two simple XSLT stylesheets were written to convert the pre-existing log file and HTML results into `testresult.xml` files that were then uploaded to eXist-db to bring the database up-to-date.

3.4. Fatal Attraction

Files with fatal errors are sometimes the most interesting tests. However, they're rather less interesting if you don't know why they failed, and are totally uninteresting if you don't know that they exist.

AHRTS compares PDF output from AH Formatter, but it has nothing to work with if AH formatter aborts with a fatal error because of a problem in its source. Reporting files with fatal errors to eXist-db and including the logs from all tests required more interdependency between Jenkins jobs and between Jenkins and eXist-db.

Firstly, the Jenkins job that runs the formatter had to be modified to save the formatter's log. Secondly, the Jenkins job that runs AHRTS was modified to add XSLT transforms that augment the AHRTS overview XML to add `compare` elements for tests with fatal errors and add `log` elements to (almost) every `compare`.

The first attempt at saving the formatter log saved the log from the entire test suite as one file and used XSLT to split the text when adding `log` elements to the overview XML. However, some of the tests generated control characters in the log – for example, the "`\f`" in `\format` is a hexadecimal character reference that pro-

[2]With 1,332 `CSS2/borders` tests and 1,119 `CSS2/tables` tests, 'manageable' is a relative term.

duces a literal Ctrl-o in the log. That could be handled by switching the XSLT processing to use XML 1.1, which allows control codes in the form `&#xf;`, etc. The literal control code wasn't a problem for the `unparsed-text()` function, but even the `unparsed-text()` function and XML 1.1 couldn't cope with an unpaired Surrogate Pair character code. To avoid a problem with one file affecting all logs, the current processing: saves the log from each test as a separate log file; prepends and appends markup to each log to make the log text be in a CDATA section in a `log` element; then accesses the XML logs from XSLT by using `collection()`. Tests with fatal errors now show up in the eXist-db app. All except a handful of test results also have the log from running the test.

Making changes to the XML before uploading it to eXist-db is a slippery slope. The same Jenkins job now also runs more XSLT to group the `compare` elements by module and to pre-compute and annotate the `compare` elements with the result of a per-`compare` calculation that was previously done on-the-fly in eXist-db.[3]

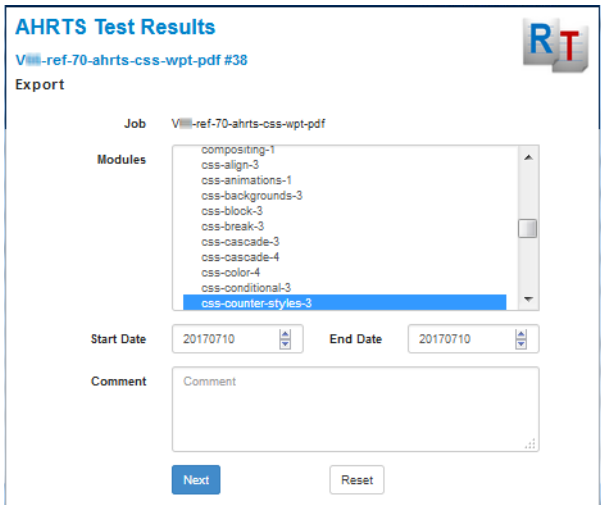
3.5. Import and Export

As stated above, my colleagues in Japan also operate their own eXist-db instance with their own copy of the app. The Japan office maintains the master copy of the test results, so it was necessary to add a way to export results from my database for import into their database. eXist-db has XQuery functions for reading and writing Zip files [7], so this was quite easy.

3.5.1. Export

The web page for selecting the module or modules to export, and the date range of results from those modules, is shown below.

[3]I had previously tried the same sort of grouping and pre-calculation after uploading by using a trigger in eXist-db [8]. The trigger worked with eXist-db on Linux but not with eXist-db on Windows, so I did not continue with it.



AHRTS Test Results

V[redacted]-ref-70-ahrts-css-wpt-pdf #38

Export

Job V[redacted]-ref-70-ahrts-css-wpt-pdf

Modules
compositing-1
css-align-3
css-animations-1
css-backgrounds-3
css-block-3
css-break-3
css-cascade-3
css-cascade-4
css-color-4
css-conditional-3
css-counter-styles-3

Start Date 20170710 End Date 20170710

Comment Comment

Next Reset

Figure 16. Export page

Clicking 'Next' takes you to a page where you can review your selection before generating the Zip file to be downloaded. eXist-db provides a `compression:zip()` function that takes a sequence of sources to zip. The sources can either be an URI referring to a resource in the database or an `entry` element for adding content to the Zip file on-the-fly. The exported Zip file contains the test results plus manifests of what is being exported so that a person can make sense of what's in each Zip file. In principle, multiple export Zip files can be unzipped into one directory with no overlap of their metadata files (other than `export.xml`) and the sum of the test results can then be uploaded manually using, e.g., eXist-db's Java client. In practice, no-one has had to do that, since the import facility has yet to cause a problem.

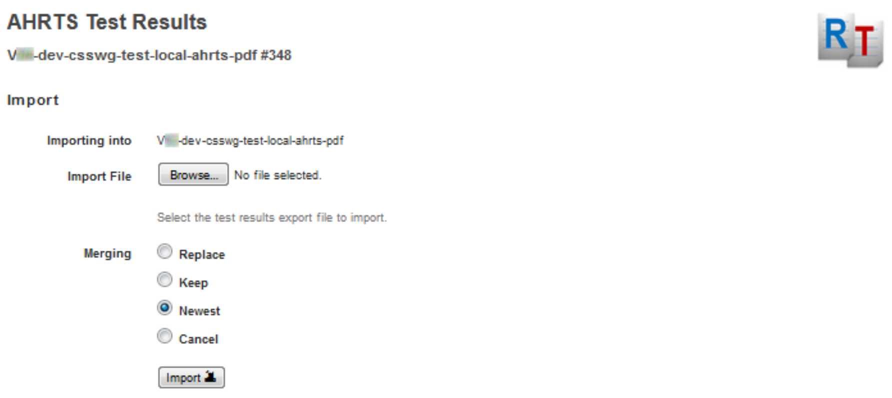
```
let $job as xs:string := request:get-parameter("job", ()),
    $start-date as xs:string :=
      request:get-parameter("start-date", ()),
    $end-date as xs:string :=
      request:get-parameter("end-date", ()),
    $modules as xs:string+ :=
      request:get-parameter("modules", ()),
    $all-modules as xs:string* :=
      request:get-parameter("all-modules", ()),
    $comment as xs:string? :=
      request:get-parameter("comment", ()),
...
    $tests-list-entry-name as xs:string :=
        concat("tests-", $basename, ".txt"),
    $tests-list-entry as element(entry) :=
        <entry name="{$tests-list-entry-name}"
                type="text" method="deflate">{
            string-join(($tests, ""), "&#xD;&#xA;")
        }</entry>,
    $export-entry as element(entry) :=
        <entry name="export.xml" type="xml" method="store">
          <export
            version="{$export-format-version}"
            job="{$job}"
            start-date="{$start-date}"
            end-date="{$end-date}"
            modules="{$modules-list-entry-name}"
            tests="{$tests-list-entry-name}"
            comment="{$comment-entry-name}"
        /></entry>,
    $testresults as xs:anyURI* :=
        for $test in $tests return
           xs:anyURI(concat($compare-output-base-uri,
                            $test,
                            '/testresult.xml')),
    $version-entry as element(entry) :=
        <entry name="version.txt" type="text"
          method="deflate">{$export-format-version}</entry>,
    $zip-name as xs:string := concat($basename, ".zip"),
    $zip := compression:zip(($export-entry,
                             $version-entry,
                             $comment-entry,
                             $modules-list-entry,
                             $tests-list-entry,
                             $testresults),
                             true(),
                             concat($ahrts-data-home, $job))
return (
        response:set-header("Content-Disposition",
                            concat("attachment; filename=",
                                   $zip-name)),
        response:stream-binary($zip, "application/zip", $zip-name)
)
```

An example `export.xml` file is below:

```
<export version="0.1" job="Vxx-ref-70-ahrts-csswg-test-pdf"
start-date="20160101" end-date="20170506"
modules="modules-Vxx-ref-70-ahrts-csswg-test-pdf-css-counter-
styles-3-20160101-20170506.txt"
tests="tests-Vxx-ref-70-ahrts-csswg-test-pdf-css-counter-styles-
3-20160101-20170506.txt"
comment="comment-Vxx-ref-70-ahrts-csswg-test-pdf-css-counter-
styles-3-20160101-20170506.txt"/>
```

3.5.2. Import

The web page for selecting an export Zip file to import is shown below:



AHRTS Test Results

V[redacted]-dev-csswg-test-local-ahrts-pdf #348

Import

Importing into	V[redacted]-dev-csswg-test-local-ahrts-pdf
Import File	Browse... No file selected.
	Select the test results export file to import.
Merging	○ Replace ○ Keep ● Newest ○ Cancel

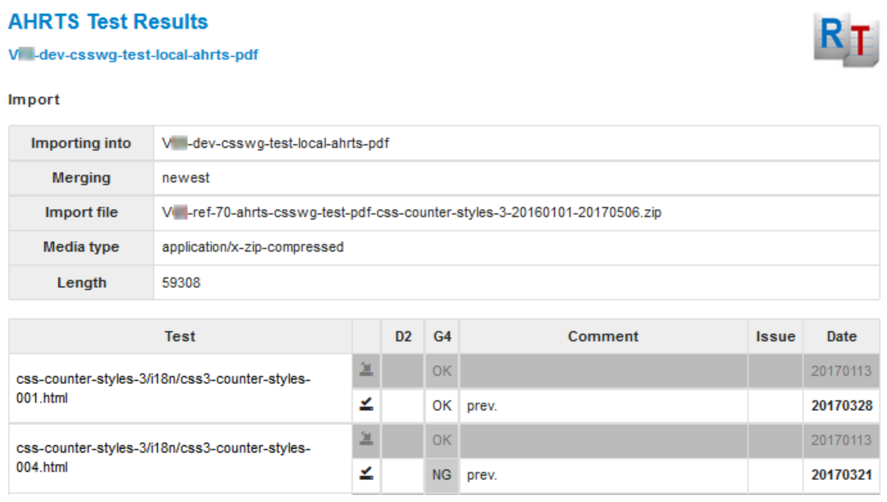
Import

Figure 17. Import form

The only difficulty with importing test results is knowing what to do when the imported data has a result for a test that already has a result in the database. The form offers four alternatives:

Replace	A result in the imported data replaces an existing result.
Keep	Do not import a result for which there is an existing result.
Newest	Use the newer of the imported or existing result for a test. When both have the date, keep the existing result.
Cancel	Import the results from the import Zip file only if it has no duplicates with existing results.

Importing a zip file also shows a summary of the imported data and the result of any merges:



AHRTS Test Results

V[redacted]-dev-csswg-test-local-ahrts-pdf

Import

Importing into	V[redacted]-dev-csswg-test-local-ahrts-pdf
Merging	newest
Import file	V[redacted]-ref-70-ahrts-csswg-test-pdf-css-counter-styles-3-20160101-20170506.zip
Media type	application/x-zip-compressed
Length	59308

Test		D2	G4	Comment	Issue	Date
css-counter-styles-3/i18n/css3-counter-styles-001.html			OK			20170113
	✓		OK	prev.		**20170328**
css-counter-styles-3/i18n/css3-counter-styles-004.html			OK			20170113
	✓		NG	prev.		**20170321**

Figure 18. Import check page (detail)

eXist-db also makes it easy to unzip files. The `compression:unzip()` function takes function arguments that are used, firstly, to filter out Zip-file entries that are not to be extracted – in this case, the `export.xml` and textual metadata files – and, secondly, to do the actual storing of extracted resources. Since the app shows a summary of the imported data and the result of any merges, this second function simply returns information about the resource. This information is to generate the table of results and is then reused to control the storing of the data.

```
declare function
local:filter($path as xs:string,
             $data-type as xs:string,
             $param as item()*) as xs:boolean {
    if ($path eq 'export.xml' or ends-with($path, '.txt'))
        then false()
    else true()
};

declare function
local:lookup($path as xs:string,
             $data-type as xs:string,
             $data as item()?,
             $param as item()*) {
    let $job as xs:string := $param[1],
        $merging as xs:string := $param[2],
        $existing-path as xs:string :=
            concat($ahrts-data-home, $job, '/', $path),
        $existing as document-node()? :=
            if (doc-available($existing-path))
                then doc($existing-path)
            else (),
        $action as xs:string :=
            if (exists($existing))
                then if ($merging eq 'replace')
                    then 'replace'
                else if ($merging eq 'keep')
                    then 'keep'
                else if ($merging eq 'newest')
                    then if ($data/testresult/@date/string() >
                             $existing/testresult/@date/string())
                             then 'newest-replace'
                         else 'newest-keep'
                else if ($merging eq 'cancel')
                    then 'conflict'
                else 'error'
            else 'new'
    return
```

```
        [ $path, $action, $data, $existing ]
};

declare function
local:import-zip($file as element(file),
                 $job as xs:string,
                 $merging as xs:string) as item()* {
    let $filter :=
          function-lookup(QName('http://www.w3.org/2005/xquery-
local-functions',
                                'filter'), 3),
        $list := function-lookup(QName('http://www.w3.org/2005/
xquery-local-functions','lookup'), 4),
        $results as array(*)* :=
           compression:unzip($file,  $filter, (), $list,
                             ($job, $merging)),
        $html := local:results-to-html($results),
        $conflict as xs:boolean :=
            some $result in $results
            satisfies $result(2) = 'conflict'
    return
    (...,
    if ($conflict)
        then local:alert("Conflicts between imported and
existing test results.  Cannot continue")
    else (<p>{local:store-results($job, $results)}</p>,
          local:success("Inserted " ||
                        count($results[?2 = 'new']) ||
                        " results."),
          local:success("Replaced " ||
                         count($results[?2 =
                            ('replace', 'newest-replace')]) ||
                         " results."))
    )
};
```

3.6. Running Jenkins from eXist-db

The eXist-db app copies a test's PDF output to the appropriate AHRTS 'base' directory using the same 'testresult' Jenkins job that was used back when updating was done using links in the AHRTS individual report PDF files. This, too, is straightforward since the eXist-db app can make a HTTP request to Jenkins to remotely trigger execution of the Jenkins job.

```
(:  Get Jenkins to copy the 'new' PDF to the 'base' PDF directory :)
declare function common:jenkins-update($job as xs:string,
                                       $test as xs:string) {
```

```
let $config-uri := concat($common:ahrts-data-home, $job,
                          '/jenkins-config.xml'),
    $config as element(jenkins-config)? :=
        if (doc-available($config-uri))
            then doc($config-uri)/jenkins-config
         else ()
    return
        if (exists($config))
            then let $external-destination :=
                        concat('http://', $config/host, ':',
                               $config/port, '/job/',
                        $config/updatejob, '/buildWithParameters'),
                     $copy-uri :=
                        concat($external-destination,
                               '?',
                               'RESULT=copy&amp;PDFDIR=',
                               encode-for-uri($config/pdfdir),
                               '&amp;TESTNAME=',
                               encode-for-uri($config/testname),
                               '&amp;EDITION=',
                               encode-for-uri($config/edition),
                               '&amp;VERSION=',
                               encode-for-uri($config/version),
                               '&amp;NEWPDF=',
                               encode-for-uri($test),
                               '.pdf'),
                     $get := httpclient:get($copy-uri, false(), ())
                 return $copy-uri
        else ()
};
```

3.7. XForms or Bootstrap?

"Wait a minute," shouted Ford Prefect. "Wait a minute!"

He leaped to his feet and demanded silence. After a while he got it, or at least the best silence he could hope for under the circumstances: the circumstances were that the bagpiper was spontaneously composing a national anthem.

"Do we have to have the piper?" demanded Ford.

"Oh yes," said the Captain, "we've given him a grant."

—The Restaurant at the End of the Universe, Douglas Adams

When developing an XML project that uses a web browser, it's hard to avoid thinking that you should use XForms. eXist-db ships with two XForms implementations, and so the first versions of the eXist-db dutifully used XForms for collecting input. However, eXist-db also ships with the Bootstrap, which is the HTML, CSS, and JavaScript framework that provides the look-and-feel of a large proportion of sites on the Web. Two of the three provided templates for an eXist-db app install Bootstrap in the new app, and, as shown in Figure 12, the current app's pages that do not need forms do use the Bootstrap-based eXist-db design. eXist-db also provides an "HTML Templating Module" [10] that makes it easy to generate HTML pages, but none of the XForms examples in the documentation use it.

Bootstrap can also style HTML forms [11]. Partly to provide a consistent 'look' for the app, but mostly because the XForms pages looked dated compared to the Bootstrap pages, all but one of the original XForms in the app have been replaced by HTML forms that are styled using Bootstrap. Can you pick which of the preceding screen shots uses an XForm?[4]

It is possible to achieve a 'mostly-Bootstrap' appearance by combining Bootstrap classes for structural elements with XForms markup for the form fields, for example:

```
<div class="form-group">
    <label class="col-xs-2 col-sm-2 control-label">Import
File</label>
    <div class="col-sm-10">
        <xf:upload id="upload1" ref="file"
            mediatype="application/x-zip-compressed"
            accept="application/x-zip-compressed">
            <xf:filename ref="@filename"/>
            <xf:mediatype ref="@mediatype"/>
            <xf:size ref="@size"/>
        </xf:upload>
        <span class="help-block" style="margin-bottom: 0">Select
the test results export file to import.</span>
    </div>
</div>
```

but the end result is still unsatisfactory since some parts of the form don't quite line up correctly and some aspects of the styling, such as the appearance of the buttons, can't be worked around, as shown in the following figure.

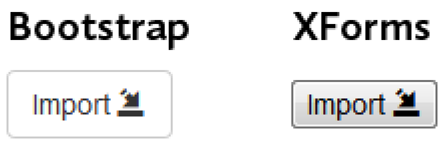


Figure 19. Bootstrap and XForms 'Import' buttons

[4]It's Figure 17. It still uses an XForm since I had trouble when submitting an uploaded file plus other parameters to eXist-db.

3.8. HTML Templating

The HTML templating mechanism [10] is separate from Bootstrap. It provides a convenient mechanism for generating HTML pages since:

- The bones of a class of pages can be provided by a single structural template HTML file.
- An individual HTML page contains the HTML markup for just the part of the template – e.g., a `div` in the `body` – that are specific to that page.
- That HTML markup can include `data-*` attributes to specify XQuery functions and function parameters. The result of an XQuery function can replace or be wrapped by the markup in the HTML file. Alternatively, the XQuery function can add values to an XQuery map that is available to all XQuery functions that are called for elements nested within the current HTML element.
- The framework also handles HTML parameters, which means less housekeeping code in your XQuery.

  The theory, from the documentation, is that:

  Ideally people should be able to look at the HTML view of an application and modify its look and feel without knowing XQuery. The application logic - written in XQuery - should be kept separate. Likewise, the XQuery developer should only deal with the minimal amount of HTML which is generated dynamically.

  My practice, however, has tended towards putting a minimum in the HTML:

```
<div xmlns="http://www.w3.org/1999/xhtml"
  data-template="templates:surround"
  data-template-with="templates/report-page.html"
  data-template-at="content">
    <div class="col-md-12" data-template="app:individual">
        <div class="row">
        <div class="col-xs-10 col-sm-10">
        <h1 data-template="app:individual-title">Generated page</h1>
</div>
<div class="col-xs-2 col-sm-2">
                <a id="poweredby" href="index.html"/>
</div>
        </div>
                <div data-template="app:individual-form"/>
                <div data-template="app:individual-nav"/>
                <div data-template="app:individual-pdf"/>
                <div data-template="app:individual-log"/>
        </div>
</div>
```

and doing more in the XQuery:

```
(: Populate $model for an individual page. :)
declare
    %templates:wrap
function app:individual($node as node(),
                        $model as map(*),
                        $job as xs:string,
                        $test as xs:string) {
    let $report as element(report)? :=
            doc(concat($app:data-home, $job, '/compareOutput/',
                       $test, '/base_vs_new.xml'))/report,
        $result := $report/analysis/result,
        $base as xs:string? :=
          $result/inputfile[@version = 'base']/string(),
        $new as xs:string? :=
          $result/inputfile[@version = 'new']/string(),
        $compare :=
          doc(concat($app:ahrts-data-home, $job,
                     '/reports/digest_vs_reports.xml'))
          /overview/compare[@name eq $test],
        $prev as xs:string? :=
          $compare/preceding-sibling::compare[1]/@name/string(),
        $next as xs:string? :=
          $compare/following-sibling::compare[1]/@name/string(),
        $pdf as xs:string? :=
          $compare/@individual-report-pdf/string(),
        $log as element(log)? := $compare/log
    return
        map { "report" := $report,
              "compare" := $compare,
              "prev" := $prev,
              "next" := $next,
              "pdf" := $pdf,
              "log" := $log }
};

(: Page title for an individual page. :)
declare function
app:individual-title($node as node(),
                     $model as map(*),
                     $job as xs:string,
                     $test as xs:string?) {
    let $report := $model("report")
    return
       (<h1 class="main-title"><a href="index.html"
>{$config:expath-descriptor/expath:title/text()}</a></h1>,
        <h3 class="report-title">
            <a href="summary.html?job={encode-for-uri($job)}">{
                if (exists($report))
                    then $report/@overview-report-title/string()
                else $job
            }</a>
               </h3>,
       if (exists($test))
         then <h2>{translate($test, '=^', '//')}</h2>
       else ())
};
```

I find that the templating mechanism works quite well and is easy to use once you get the hang of it. One problem, however, is that the keys of the map entries are not validated, so a typo where the key is defined or anywhere where it is referenced can lead to a mysterious empty sequence simply because the keys don't match.

3.9. '=' in file names

AHRTS can currently only compare files in two directories, and it does not look into subdirectories. Most test suites – including the CSS test suite – have files arranged in subdirectories. To get around this difference, the Jenkins job that runs the formatter would write the PDFs to file names where '=' (which does not appear in any test file names) was used in place of the directory separator. This worked fine before eXist-db was used. However, '=' needs to be escaped in parameters in URLs, and using '=' in file names exposed multiple bugs in eXist-db. For example, collections with names containing '=' can't be opened in the eXide editor's 'Manage' interface, and the only way found for deleting them is by using eXist-db's Java interface.

To their credit, the eXist-db developers responded quickly to the initial bug reports, but fixing them all will take time. It was simply more reliable to just use a different separator character sequence, but doing that required a lot of renaming of files on Jenkins's file system and renaming resources in eXist-db. Since the PDF files with '=' in their file names couldn't be renamed programmatically, the collection that contained them had to be deleted and a whole new set of PDFs uploaded.

3.10. CSS Tests now Web Platform Tests

In the first half of 2017, the CSS Working Group migrated their tests from their own GitHub repository to being under a subdirectory of the Web Platform Tests project. Several of the modules were renamed during the migration, so it was necessary to migrate the corresponding test results to new URIs to match.

It is unlikely that export files from before the changeover will ever be needed again but, just to make sure that nothing is lost, the code for importing test results now handles both old file names containing '=' and old, pre-WPT module names and maps them to current usage.

3.11. Localization

The practical upshot of this is that if you stick a Babel fish in your ear you can instantly understand anything said to you in any form of language.
—Hitchhiker's Guide to the Galaxy, Douglas Adams

The current 'app' is almost completely localized for both English and Japanese. Some localization into English was necessary because the 'D2' states were initially only provided as Japanese text. Some of the localizations into Japanese were rolled back at the request of my colleagues in Japan because they found the English easier to understand than the notionally equivalent nouns and verbs that I plucked from an online English–Japanese dictionary.

eXist-db has a localization library [12] that uses files with a format that is very similar to Java XML property files.

```
<catalogue xml:lang="ja">
  <msg key="Comment">コメント</msg>
  <msg key="Date">日</msg>
  <msg key="Diff">相違</msg>
  <msg key="Issue">発行</msg>
  <msg key="Test">テスト</msg>
</catalogue>
```

Localizations are applied using `i18n:text` elements in a mechanism similar to the HTML templating mechanism:

```
<tr>
  <th><i18n:text key="Test">Test</i18n:text></th>
  <th><i18n:text key="Diff">Diff</i18n:text></th>
  <th>D2</th>
  <th>G4</th>
  <th><i18n:text key="Comment">Comment</i18n:text></th>
  <th><i18n:text key="Issue">Issue</i18n:text></th>
  <th><i18n:text key="Date">Date</i18n:text></th>
</tr>
```

This works well enough, but language selection (in my opinion) is not straightforward, and the standard library does not provide the option of selecting the language from the browser's `Accept-Language` header. At present, the app uses its own version of the i18n library that is based on a version [13] by Wolfgang Meier, one of the eXist-db developers. This version can use either the `Accept-Language` header or a language setting configured in the app.

Using elements to handle localization isn't helpful for localizing attribute values. The functions provided for use from XQuery require specifying both the path to the localization files and the current language, plus repeating the text in the `i18n:text` element's content and its `key` attribute seemed redundant. I made some convenience functions that: wrap the regular i18n processing; operate on both text and attribute values; get the localization files' path and language from the app's configuration; and require only one copy of the text being localized. For example:

```
<input type="text" class="form-control" name="issue" id="issue"
  placeholder="{common:i18n-text('Issue numbers')}"
  value="{$testresult/issue/string()}" accesskey="i"/>
```

No method has yet been found to localize the messages popped up by the BetterForms XForms implementation.

3.12. Dashboard

As stated previously, the mass of tests is divided into modules to make the work more manageable. Information about the modules and their relative priorities was initially maintained as a wiki page, but that was later migrated to a spreadsheet. When my colleagues in Japan wanted to also see the priorities in the eXist-db app, I both added a mechanism to paste tab-delimited text from the priorities spreadsheet into the app and provided a dashboard summarizing the results for each module and each priority level.

Over time, however, my colleagues in Japan have requested additional information on the dashboard for combinations of test result status values that are useful to them, as shown in Figure 20.

And despite their being the impetus for dividing the summary view by module, they also requested views of all tests with particular status combinations, as shown in Figure 21.

4. Variations

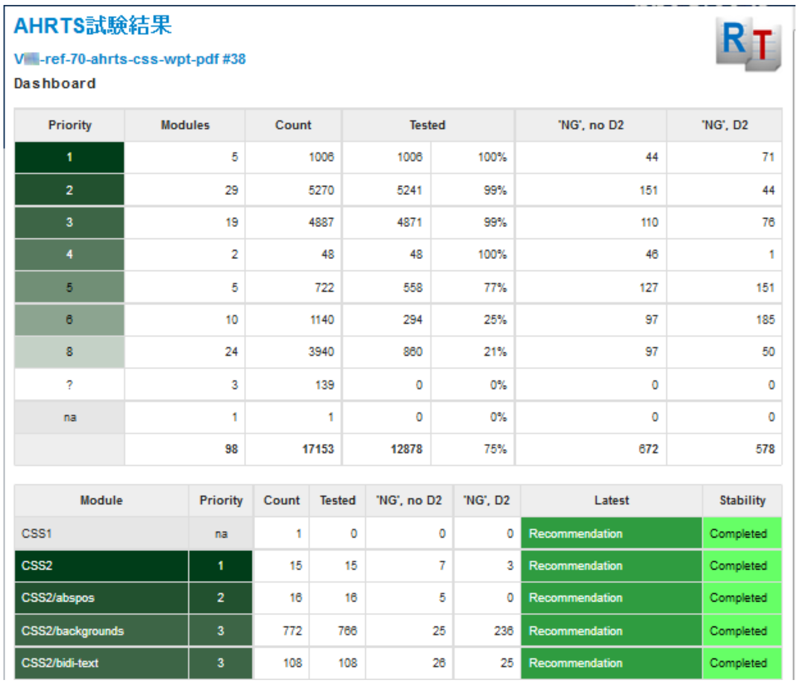
One of the troublesome circumstances was the Plural nature of this Galactic Sector, where the possible continually interfered with the probable.

—*Mostly Harmless, Douglas Adams*

Two local variations have been developed: testing a local fork of the AH Formatter code and checking XSL-FO test results.

4.1. Testing an AH Formatter fork

When a local fork of the AH Formatter code has changes that are not yet in the main AH Formatter code base, the local fork can produce output that is different



AHRTS試験結果

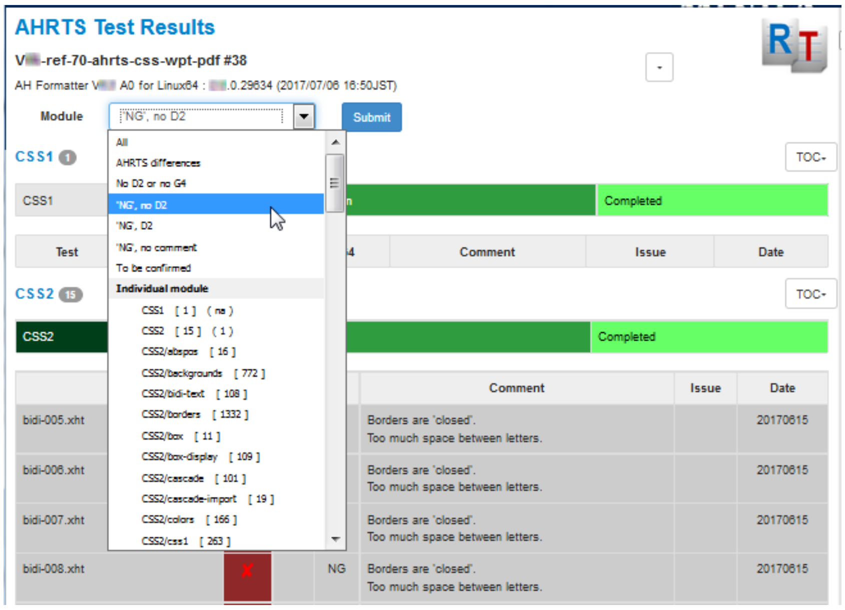
V[redacted]-ref-70-ahrts-css-wpt-pdf #38

Dashboard

Priority	Modules	Count	Tested		'NG', no D2	'NG', D2
1	5	1006	1006	100%	44	71
2	29	5270	5241	99%	151	44
3	19	4887	4871	99%	110	76
4	2	48	48	100%	46	1
5	5	722	558	77%	127	151
6	10	1140	294	25%	97	185
8	24	3940	860	21%	97	50
?	3	139	0	0%	0	0
na	1	1	0	0%	0	0
	98	17153	12878	75%	672	578

Module	Priority	Count	Tested	'NG', no D2	'NG', D2	Latest	Stability
CSS1	na	1	0	0	0	Recommendation	Completed
CSS2	1	15	15	7	3	Recommendation	Completed
CSS2/abspos	2	16	16	5	0	Recommendation	Completed
CSS2/backgrounds	3	772	766	25	236	Recommendation	Completed
CSS2/bidi-text	3	108	108	26	25	Recommendation	Completed

Figure 20. Dashboard



AHRTS Test Results

V[redacted]-ref-70-ahrts-css-wpt-pdf #38

AH Formatter V[redacted] A0 for Linux64 : [redacted].0.29634 (2017/07/06 16:50JST)

Module: 'NG', no D2 — Submit

Module options:
- All
- AHRTS differences
- No D2 or no G4
- 'NG', no D2
- 'NG', D2
- 'NG', no comment
- To be confirmed
- **Individual module**
  - CSS1 [1] (na)
  - CSS2 [15] (1)
  - CSS2/abspos [16]
  - CSS2/backgrounds [772]
  - CSS2/bidi-text [108]
  - CSS2/borders [1332]
  - CSS2/box [11]
  - CSS2/box-display [109]
  - CSS2/cascade [101]
  - CSS2/cascade-import [19]
  - CSS2/colors [166]
  - CSS2/css1 [263]

CSS1 1 — TOC

CSS1	n	Completed

Test	4	Comment	Issue	Date

CSS2 15 — TOC

CSS2		Completed

			Comment	Issue	Date
bidi-005.xht			Borders are 'closed'. Too much space between letters.		20170615
bidi-006.xht			Borders are 'closed'. Too much space between letters.		20170615
bidi-007.xht			Borders are 'closed'. Too much space between letters.		20170615
bidi-008.xht	X	NG	Borders are 'closed'. Too much space between letters.		20170615

Figure 21. New summary views

from the output of the main AH Formatter code base. The output can also change as changes are made in the local fork. The current output from the fork need to be checked against both the previous output from the fork and the current output from the main code.

Each job in the eXist-db app can be configured to refer to a 'reference' job. The individual result pages for each test in that job will also show the AHRTS individual report PDF for the corresponding test in the 'reference' job. This allows the results from testing the fork to be compared against the corresponding results from testing the main code. It also allows the results from testing the main code to be compared against the results from testing the current public release of AH Formatter.

4.2. Testing XSL-FO

Using the combination of Jenkins, AHRTS, and eXist-db to run AH Formatter on XSL-FO tests and make the results available for review is almost the same as when testing CSS. The only differences are:

- The XSL-FO tests do not have the same 'module' structure.

  The XSL-FO tests are arranged into `demodata` and `testdata` directories that each have multiple subdirectories under them. The Jenkins configuration for jobs that run XSL-FO tests simply use a different XSLT file when grouping the `compare` elements by module. The XSLT file imports the same XSLT files that are used when processing CSS test results and overrides the single XSLT function that handles the grouping into modules.

  ```
  <xsl:import href="ahrts-prep-for-xmldb.xsl" />

  <!-- ahf:compare-module($compare as element(compare)) as xs:string?
       Returns the module name to use for $compare.  Used when
       grouping <compare> into <module>. -->
  <xsl:function name="ahf:compare-module" as="xs:string?">
    <xsl:param name="compare" as="element(compare)" />

    <xsl:sequence
        select="string-join(tokenize($compare/@name,
                            $directory-separator-regex)
                                    [position() &lt;= 2],
                  $directory-separator)" />
  </xsl:function>
  ```

- XSL-FO test results are not categorized by stability level.

  The per-job setting in the eXist-db app allow a job to be configured as an 'XSL-FO' or 'CSS' job. The summary and dashboard pages for XSL-FO jobs do not show the stability information that is shown for CSS jobs.

5. Conclusion

So long, and thanks for all the fish.

—*So Long, and Thanks for All the Fish, Douglas Adams*

Feedback from colleagues in Japan has been uniformly positive. The effort required to correct problems and fill in comments is about the same as for the previous HTML form, but the advantages that they stated include:

- Using AHRTS for automatically identifying changed test results is a big advantage.
- Managing the tests is much more effective than with the HTML report.
- Having the test result status, comment, AHRTS report, and AH Formatter log on one page is useful.
- The extra functionality added during the course of the project has made it even more useful.
- The system has saved time and effort.

Developing a system for checking the results from 17,000 CSS tests has had a few twists and turns, but the current implementation as an eXist-db app fits the requirements as they have developed over time, is proving useful, and has made the task much easier.

Bibliography

[1] https://lists.w3.org/Archives/Public/public-css-testsuite/2010Aug/0020.html, Implementation Report Template for CSS2.1 Test Suite

[2] https://www.antennahouse.com/antenna1/antenna-house-regression-testing-system/, Antenna House Regression Testing System

[3] https://www.w3.org/Style/XSL/TestSuite/tools/testsuite.dtd, XSL 1.0 Test Suite DTD

[4] https://www.w3.org/Style/XSL/TestSuite/index.html, XSL 1.0 Test Suite

[5] http://exist-db.org/exist/apps/homepage/index.html, eXist-db

[6] http://exist-db.org/exist/apps/doc/xforms.xml, eXist-db 'XForms Introduction'

[7] http://exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/compression&location=java:org.exist.xquery.modules.compression.CompressionModule, eXist-db Compression module

[8] http://exist-db.org/exist/apps/doc/triggers.xml, eXist-db "Configuring Database Triggers"

[9] http://getbootstrap.com/, Bootstrap

[10] http://exist-db.org/exist/apps/doc/templating.xml, eXist-db "HTML Templating Module"

[11] http://getbootstrap.com/css/#forms, Bootstrap Forms

[12] http://exist-db.org/exist/apps/demo/examples/special/i18n-docs.html, eXist i18n XQuery Module Documentation

[13] http://markmail.org/message/l7x6bfyyg3ohwlna, Re: [Exist-open] I18n and 'Accept-Language' header?

Form, and Content

Data-Driven Forms

Steven Pemberton
CWI, Amsterdam
`<steven.pemberton@cwi.nl>`

Abstract

Because of the legacy of paper-based forms, modern computer-based forms are often seen as static data-collection applications, with rows of rectangular boxes for collecting specific pieces of data. However, they have far more opportunities for being dynamic, checking data for consistency, leaving out fields for non-relevant data, and changing structure and detail to match the data-filling flow. Furthermore, data is no longer limited to pure textual input, but can be entered using any method that is available on a computer.

While classically it is the form that drives the data produced, this paper examines how forms can be data-driven, for structure, for presentation, and for execution, and proposes that our view of forms have been severely impaired by the paper-based legacy.

Keywords: XML, XForms, Forms, Data-driven

1. Introduction

Throughout history, when new technologies have been introduced, there has been a tendency for them to imitate the old technologies they are repalcing before they iterate to their proper embodiment.

For instance, the first printed books looked like hand-written manuscripts, by using a typeface that imitated handwritten text, making them much harder to read than necessary; the first cars looked like horse-drawn carts without the horse, partly because that was what people could make at the time, but also because horse-drawn carts were what they were seen to be replacing — looking back from a modern perpective it is astonishing how long it took for anyone to have the bright idea of actually enclosing the driver in a space protected from the elements; and modern computer applications (such as agendas) often imitate their real-life counterparts in excruciating detail.

Digital forms have in a similar way long been held back by the history of paper-based form-filling. In the early days of the Web, in this author's experience, many managers required their online forms to be identical to their paper-based equivalents, even though this meant not being able to use facilities that would otherwise have made them much easier to use.

As online production and usage matures, so has the online form-filling experience improved. There are now a small number of standardised form development languages. However, form development is often still largely based around the static idea of the form filling experience, for example [2].

It is time to move up a level of abstraction. Forms are about data collection, and traditionally the shape of the form drives the shape of the data. This paper investigates how the data can drive the form, and shows that things that we might not traditionally see as forms can be viewed as data collection.

1.1. XForms

XForms [4], [5] is an XML-based markup language, originally designed only for traditional-style forms, but in later iterations generalised to more widely-applicable usage.

A principle feature of the language is its separation of data and associated data description from the actual controls used to display and enter the data. This separation of concerns can be compared to the separation of content and styling done with style sheets, and has similar advantages, making the data more tractable, and facilitating reuse.

The data in XForms is contained in a *model* that consists of any number of XML instances, which can be loaded from external sources, along with descriptions of properties and relationships that nodes in the data may have.

Controls in the user interface are then bound to data nodes using XPath expressions [3]. For instance:

```
<input ref="cc-number" label="Credit card number"/>
```

An example data property is *relevance*. As a simple example, the credit-card number to be input can be marked in the model as being relevant only if the method of payment is by credit card:

```
<bind ref="cc-number" relevant="../payment-method = 'credit'"/>
```

Controls bound to values that are not relevant, as well as controls bound to elements that are simply not present in the instance data, are *disabled*: they are not visible to the user, and they can't be used for input. As will be seen, this is an essential element of data-driven forms.

XForms is a W3C standard; a new version, 2.0, is in preparation [6], and some facilities of XForms 2.0 are used in the examples that follow.

This paper presents three case studies of the use of data-driven forms: one of data-driven structure, one of data-driven presentation, and the last of data-driven execution.

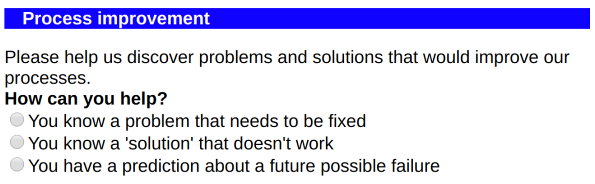
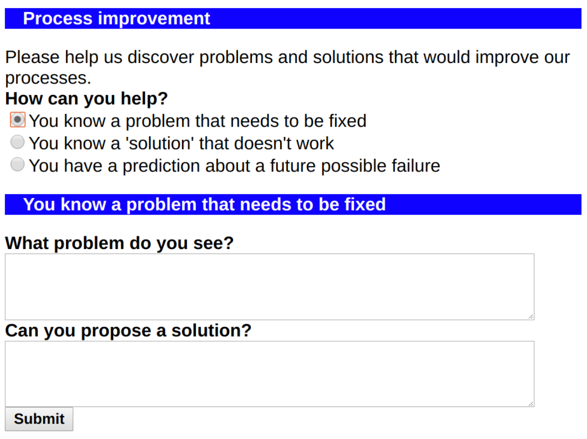
1.2. What is a Form?

A traditional form is no more than a method of data-collection, taking input from a user and storing it somewhere. Computer-based forms however are becoming steadily more dynamic, for instance, partially filling in an address based on the entry of a postcode, or calculating the final amount based on what you have ordered, the taxes, and the delivery costs. Clearly, modern forms deal with input from a user, but do calculation and output as well. Is a log-in dialogue box a form? Yes, it is. Is a widget asking you to move a pointer on a map to indicate the location you want to share a form? It can be so construed. As forms become more and more dynamic, the distinction between 'form' and 'application' becomes steadily more nebulous. This paper consequently uses a liberal definition of what constitutes a form.

2. Data-driven Structure: A Questionnaire

The first case study is a classic data-collection form. It is based on one produced for an Internet community, for identifying potential improvements for the internet. It has a short introduction, gives the user a choice of three options, and then reveals a small number of questions, based on the choice. (The actual production form was more complex, but for the sake of exposition, it has been compressed here to the essence).

Something like this:

Before making choice	After making choice
 **Process improvement** Please help us discover problems and solutions that would improve our processes. **How can you help?** ○ You know a problem that needs to be fixed ○ You know a 'solution' that doesn't work ○ You have a prediction about a future possible failure	 **Process improvement** Please help us discover problems and solutions that would improve our processes. **How can you help?** [✓ You know a problem that needs to be fixed] ○ You know a 'solution' that doesn't work ○ You have a prediction about a future possible failure **You know a problem that needs to be fixed** **What problem do you see?** **Can you propose a solution?** Submit

2.1. The Static Version

Since this is a classic style of form, in the most obvious and direct —static— approach, the controls could look like this, where the user makes a choice:

```
   Please help us discover problems and solutions that would improve our processes.
```

```
<select1 ref="choice">
   <label>How can you help?</label>
   <item><label>You know a problem that needs to be fixed</label>
         <value>problem</value></item>
   <item><label>You know a 'solution' that doesn't work</label>
         <value>failure</value></item>
   <item><label>You have a prediction about a future possible failure</label>
         <value>prediction</value></item>
</select1>
```

and as a result of the choice, one of the options is displayed:

```
<switch ref="choice">

   <case name=""/> <!-- Until a choice is made, nothing is displayed -->

   <case name="problem">
      <group ref="problem">
         <label>You know a problem that needs to be fixed</label>
         <textarea ref="problem"><label>What problem do you see?</label></textarea>
         <textarea ref="solution"><label>Can you propose a solution?</label></textarea>
      </group>
   </case>

   <case name="failure">
      <group ref="failure">
         <label>You know a 'solution' that doesn't work</label>
         <textarea ref="problem">
            <label>What 'solution' will fail or cause trouble?</label>
         </textarea>
         <textarea ref="solution"><label>Can you propose a fix?</label></textarea>
      </group>
   </case>

   <case name="prediction">
      <group ref="prediction">
         <label>You have a prediction about a future possible failure</label>
         <textarea ref="problem"><label>What is your scenario?</label></textarea>
         <select1 ref="likely">
            <label>How likely is this scenario?</label>
            <item><label>High</label><value>1</value></item>
            <item><label>Medium</label><value>2</value></item>
            <item><label>Low</label><value>3</value></item>
         </select1>
         <textarea ref="who"><label>Who should address these issues?</label></textarea>
         <textarea ref="solution"><label>Can you propose a solution?</label></textarea>
      </group>
   </case>
</switch>

<submit><label>Submit</label></submit>
```

This uses the following structure for the data; for any one answer, only the values for the selected case would get filled in:

```
<data>
   <choice/>
   <problem>
```

```
      <problem/>
      <solution/>
   </problem>
   <failure>
      <problem/>
      <solution/>
   </failure>
   <prediction>
      <problem/>
      <likely/>
      <who/>
      <solution/>
   </prediction>
</data>
```

2.2. Making the Form Multi-lingual

One of the early decisions was to make the form multi-lingual. This involved creating an instance that contained all the labels and other texts:

```
<instance id="m">
  <messages xmlns="" lang="en">
    <intro>Please help us discover problems and solutions that would
           improve our processes.</intro>
    ...
  </messages>
</instance>
```

and in the body of the form:

```
<output ref="instance('m')/intro"/>
```

For the `select1`, the messages:

```
<choice>
   <label>How can you help?</label>
   <item value="problem">You know a problem that needs to be fixed</item>
   <item value="failure">You know a 'solution' that doesn't work</item>
   <item value="prediction">You have a prediction about
                            a future possible failure</item>
</choice>
```

with the control now reading:

```
<select1 ref="choice">
 <label ref="instance('m')/choice/label"/>
   <itemset ref="instance('m')/choice/item">
       <label ref="."/>
       <value ref="@value"/>
```

```
      </itemset>
   </select1>
```

And for the groups within the cases, the messages:

```
   <problem>
      <label>You know a problem that needs to be fixed</label>
      <problem>What problem do you see?</problem>
      <solution>Can you propose a solution?</solution>
   </problem>
```

And the controls:

```
   <group ref="problem">
      <label ref="instance('m')/problem/label"/>
      <textarea ref="problem">
        <label ref="instance('m')/problem/problem"/></textarea>
      <textarea ref="solution">
        <label ref="instance('m')/problem/solution"/></textarea>
   </group>
```

and similar for the other two cases.

2.3. Generalising

As a result of this change, it was obvious how similar the three cases were. The third has a little extra detail, but otherwise they are nearly identical. If the data is changed to reflect these similarities, like this:

```
   <data>
      <choice/>
      <answer choice="problem">
         <problem/>
         <solution/>
      </answer>
      <answer choice="failure">
         <problem/>
         <solution/>
      </answer>
      <answer choice="prediction">
         <problem/>
         <likely/>
         <who/>
         <solution/>
      </answer>
   </data>
```

then the whole `switch` in the form can be replaced with a single `group` that is driven by the data. The `group` selects only the answer whose `choice` attribute matches that of the value actually chosen, and so covers all three cases:

```
<group ref="answer[@choice=../choice]">
   <label ref="instance('m')/answer[@choice=context()/@choice]/label"/>
   <textarea ref="problem">
      <label ref="instance('m')/answer[@choice=context()/../@choice]/problem"/>
   </textarea>
   <select1 ref="likely">
      <label ref="instance('m')/answer[@choice=context()/../@choice]/likely/label"/>
      <itemset ref="instance('m')/answer[@choice=context()/../@choice]/likely/item">
         <label ref="."/><value ref="@value"/>
      </itemset>
   </select1>
   <textarea ref="who">
      <label ref="instance('m')/answer@choice=context()/../@choice]/who"/>
   </textarea>
   <textarea ref="solution">
      <label ref="instance('m')/answer[@choice=context()/../@choice]/solution"/>
   </textarea>
</group>
```

Note that:

- Just as before, since initially no answer has a `choice` with a value that has been selected (since nothing has yet been selected), the `ref` will select no nodes, and so nothing will be displayed for the answers.
- If the instance data for the selected answer doesn't have a `likely` or `who` element, the controls for those elements won't be displayed, since similarly they are not bound to any node.

This change to the data requires a similar change to the structure of the messages document. Note how closely its structure mirrors that of the data:

Data	Messages
`<data>`	`<messages lang="en">`
	`   <label>Process improvement</label>`
	`   <intro>Please help us discover problems and solutions`
	`              that would improve our processes.</intro>`
`   <choice/>`	`   <choice>How can you help?</choice>`
`   <answer choice="problem">`	**`   <answer choice="problem">`**
	`     <label>You know a problem that needs to be fixed</label>`
`      <problem/>`	`     <problem>What problem do you see?</problem>`
`      <solution/>`	`     <solution>Can you propose a solution?</solution>`
`   </answer>`	`   </answer>`
`   <answer choice="failure">`	**`   <answer choice="failure">`**
	`     <label>You know a 'solution' that doesn't work</label>`
`      <problem/>`	`     <problem>What 'solutions' will fail or`
	`             cause trouble?</problem>`
`      <solution/>`	`     <solution>Can you propose a solution?</solution>`
`   </answer>`	`   </answer>`
`   <answer choice="prediction">`	**`   <answer choice="prediction">`**
	`     <label>You have a prediction about a future`
	`           possible failure</label>`
`      <problem/>`	`     <problem>What is your scenario?</problem>`
`      <likely/>`	`     <likely>`
	`       <label>How likely is this scenario?</label>`
	`       <item value="1">High</item>`
	`       <item value="2">Medium</item>`
	`       <item value="3">Low</item>`
	`     </likely>`
`      <who/>`	`     <who>Who should address these issues?</who>`
`      <solution/>`	`     <solution>Can you propose a solution?</solution>`
`   </answer>`	`   </answer>`
`</data>`	`</messages>`

The message for the group's label is selected using `answer[@choice=context()/@choice]`. This selects the `answer` element in the messages, whose `choice` attribute matches that of the context element. In this case the context element is the `answer` element in the data that has been selected.

For the first `textarea` element, the context item is now the element `problem`, which is a child of `answer`, so you have to go up one level to get to `answer`, in order to get its `choice` attribute: `item[@choice=context()/../@choice]`.

2.4. Analysis

The form is now driven from two data files: the template for the data, and the messages. Essentially, the group of controls acts as an interpreter of the data.

To illustrate this, suppose a fourth option were to be added to the form; all that is necessary is to add it to the data template, with a suitable new choice value:

```
<answer choice="solution"><problem/><who/><solution/></answer>
```

and matching messages in the message file:

```
<answer choice="solution">
   <label>You know an existing solution that can be adopted</label>
```

```
    <problem>What solution do you know of?</problem>
    <who>Who should we approach?</who>
    <solution>How could we best adopt the solution?</solution>
</answer>
```

and the form now works *without change* with the new entry.

This makes life much easier for content providers: they can make textual changes to a form without having to ask the programmers to do it, they can add new cases themselves fairly easily. In fact, you could even make a form to simplify the process!

To adapt the form for a different language, you only have to supply a translation of the message document for the new language, and add code to switch between languages by loading in the various message documents.

3. Data-driven Display: A Presentation Manager

The CSS [1] styling language has a special *presentation* mode. If you include a set of rules grouped as *projection* media, like so:

```
@media projection {
  ...
}
```

then when the browser is put into presentation mode, those styling rules apply. The idea is that the browser goes into full-screen mode, and the projection rules will typically increase the font size, and express where 'page' breaks are. As a result, this allows you to avoid using proprietary presentation software, and use HTML, plus CSS with a projection mode, and present from a browser.

This has had several advantages, for instance:

- it makes the content easily repurposable,
- it gives a choice of editing software,
- it is platform independent,
- it gives a lot of control,
- but most important: it guarantees a long life for the content. The use of proprietary software always brings with it the risk of the content no longer being readable after several years.

Unfortunately, and shamefully, only one browser ended up supporting presentation mode, Opera, but now even Opera has discontinued support.

Since the effective demise of presentation mode, many packages of Javascript have emerged to support presentation in combination with HTML5, such as [9], [10], [11], [12], and [13] (and *dozens* more) and although some of them are very cute, they all have some underlying problems:

- They are largely not standardised: each has its own format for slides, and its own package of javascript, so you can't swap between packages;
- If you want to repurpose existing content, you have to edit the content files;
- If support for the package disappears (which has already happened for some packages), you are in trouble: this is comparable to the problem of using proprietary software.

To mitigate these problems, one solution is to use XForms to display the slides. Of course, this is not quite as good as having a standard built into the browser, but the advantages include:

- It is very easy: it is a surprisingly small amount of markup;
- it allows the continued use and repurposing of existing content without change;
- it continues to give the power of XHTML+CSS for styling.

3.1. Slide Deck

Each slide deck is an XHTML document, where, in this example, each slide is a top-level `div` containing XHTML including images.

The initial slide deck is loaded into an XForms instance like this:

```
<instance id="slides"
          src="http://www.cwi.nl/~steven/Talks/2018/prague/"/>
```

(we'll see later how to load different decks).

The central part of the application is then an XForms `group` that handles a single `div`:

```
<group ref="h:body/h:div[position()=instance('i')/index]">
   ...
</group>
```

This selector defines how to find a single slide within the instance, so it can be considered cleaner to gather the instance and this definition together:

```
<instance id="slides"
          src="http://www.cwi.nl/~steven/Talks/2018/prague/"/>
<bind id="slide" ref="h:body/h:div"/>
```

and then use this for the group. In this way, the controls in the form are independent of the data:

```
<group ref="bind('slide')[position()=instance('i')/index]">
   ...
</group>
```

(The `bind` function is an XForms 2.0 feature)

Either way, this requires an administration instance to keep track of which slide is visible at any time, initalised to 1:

```
<instance id="i">
    <admin xmlns="">
        <index>1</index>
    </admin>
</instance>
```

Although buttons *could* be added to step through the slides like this:

```
<trigger label="←">
    <setvalue ev:event="DOMActivate"
              ref="instance('i')/index" value=". - 1"/>
</trigger>
<trigger label="→">
    <setvalue ev:event="DOMActivate"
              ref="instance('i')/index" value=". + 1"/>
</trigger>
```

it is preferable to do it via the keyboard, not least because presentation remotes act as if they are keyboards, sending the characters "`Page Up`" and "`Page Down`" when the buttons are pressed:

```
<action ev:event="keydown" ev:defaultAction="cancel">
    <setvalue ref="instance('i')/index"
              if="event('key')='PageUp'
                  or event('key')='ArrowLeft'"  value=". - 1"/>
    <setvalue ref="instance('i')/index"
              if="event('key')='PageDown' or
                  event('key')='ArrowRight'" value=". + 1"/>
</action>
```

(It is necessary to cancel the default action of the event, since otherwise the browser would do a page up or down as well.)

3.2. Displaying One Slide

Now that we have the infrastructure to step through each slide, we can define how an individual slide should be presented.

Each slide contains a sequence of XHTML elements. So within the `group` holding the slide, each of those elements have to be displayed. They are treated one by one within a `repeat`:

```
<repeat ref="*">
   ...
</repeat>
```

Here are some simple cases:

```
<output class="h1" ref=".[name(.)='h1']"/>
<output class="h2" ref=".[name(.)='h2']"/>
<output class="pre" ref=".[name(.)='pre']"/>
```

The XPath idiom "`.[name(.)='h1']`" selects the current element only if its name is 'h1'. If its name doesn't match, then no node is selected by the output element, and so the control is *disabled* and is not rendered; if the name matches, then its content is output. By attaching a class, CSS controls how it will be displayed. Clearly at most one of the output elements will be enabled.

In fact these can be combined into one output element by taking advantage of XForms 2.0 attribute value templates:

```
<output class="{name(.)}"
        ref=".[name(.)='h1' or name(.)='h2' or name(.)='pre']"/>
```

More complicated cases are those elements that themselves contain other elements, such as `<p>` and `<ul>`.

The easier of these two is `<ul>`. Here a similar trick is used, with a repeat over the contained elements, with the advantage that we know they are all `<li>` elements:

```
<group class="ul" ref=".[name(.)='ul']">
    <repeat ref=".[name(.)='li']">
        <output class="li" ref="."/>
    </repeat>
</group>
```

The `<p>` elements have a complication that they may contain mixed content. For this, rather than using the selector "*", the selector "node()" is used, which selects all child nodes: text and comments as well as elements:

```
<group class="p" ref=".[name(.)='p']">
    <repeat ref="node()">
        <output class="text" ref=".[name(.)='#text']"/>
        <output class="{name(.)}"
                ref=".[name(.)='em' or name(.)='strong' or ▶
name(.)='code' or name(.)='a']"/>
        <output class="img"
                ref=".[name(.)='img']" value="concat(instance('i')/base, ▶
@src)" mediatype="image/*"/>
    </repeat>
</group>
```

Since there is no output element that selects comment nodes, they won't be displayed.

The only interesting case here is for images. The `src` attribute is relative to the original slides, so must be concatenated with the base URL of the slides, which can be stored in the admin instance:

```
<instance id="i">
    <admin xmlns="">
        <index>1</index>
        <base>https://homepages.cwi.nl/~steven/Talks/2018/prague/</base>
    </admin>
</instance>
```

3.3. Loading Other Slide Sets

Having the base stored in the admin instance makes it easy to load another slide set. The user supplies the URL of the new slide set, it gets submitted, and the result is used to replace the slides instance:

```
<input ref="instance('i')/base" label="URL:"/>
<submit submission="change" label="Go"/>
```

where the `<submission>` element looks like this, remembering also to set the index back to 1:

```
<submission id="change" resource="{instance('i')/base}"
            method="get" serialize="none"
            replace="instance" instance="slides">
    <action ev:event="xforms-submit-done">
        <setvalue ref="instance('i')/index" value="1"/>
    </action>
</submission>
```

3.4. Analysis

The input from the user for this form is minimal: it is a URL for a slide set, and a single integer, indicating which slide to display, which is incremented and decremented via keystrokes, or emulated keystrokes from a presentation remote. Although it was presented as an 'administrative' value, it is in fact the central piece of input. The controls, in a similar way to the first example, are an interpreter for the data in the selected slide; the result can be considered a 'presentation' of the integer.

4. Data-driven Control: The XForms 2.0 Test Suite

XForms 1.0 and 1.1 both had test suites that consisted largely of static XForms documents [7], [8]. If you wanted to add more cases to a test, it involved adding to the set of documents, or editing the individual documents.

The test suite for XForms 2.0 now being constructed takes a different approach. While different parts of the test suite have different structures, depending on what is being tested, we consider here the testing of functions.

4.1. Testing Functions

It is required to test that functions like

```
compare('apple', 'orange')
```

return the right result.

To do this, the string is enclosed in an element:

```
<test>compare('apple', 'orange')</test>
```

sub-elements are added to identify the parameters

```
<test>compare('<a>apple</a>', '<b>orange</b>')</test>
```

and attributes added to store the required result, the actual result, and whether the test case passes or not:

```
<test pass="" res=""
       req="-1">compare('<a>apple</a>', '<b>orange</b>')</test>
```

As many such test cases as necessary are then gathered together in an instance:

```
<instance>
    <tests pass="" name="compare() function" xmlns="">
        <test pass="" res=""
              req="-1">compare(<a>apple</a>, <b>orange</b>)</test>
        <test pass="" res=""
              req="1">compare(<a>orange</a>, <b>apple</b>)</test>
        <test pass="" res=""
              req="0">compare(<a>apple</a>, <b>apple</b>)</test>
        ...
    </tests>
</instance>
```

A bind is then used to calculate the individual results:

```
<bind ref="test/@res" calculate="compare(../a, ../b)"/>
```

another bind, independent of which function is being tested, decides if each test case has passed:

```
<bind ref="test/@pass" calculate="if(../@res = ../@req, 'yes', 'no')"/>
```

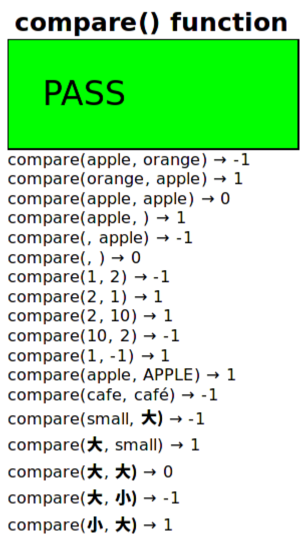
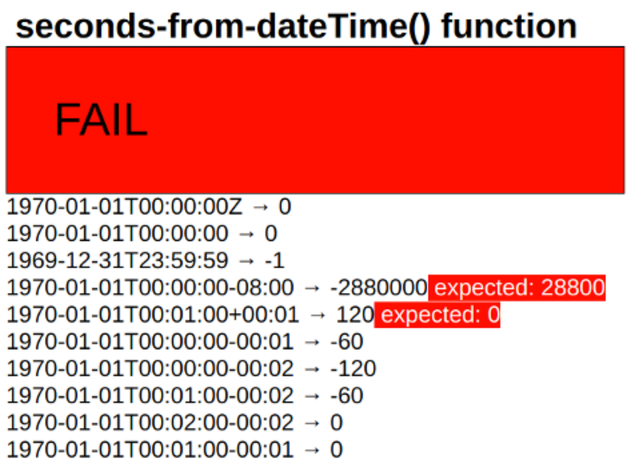
and finally a bind for the attribute on the outmost element records if all tests have passed:

```
<bind ref="@pass" calculate="if(count(//test[@pass!
='yes'])=0, 'PASS', 'FAIL')"/>
```

With this structure, every test form has an identical set of controls, that output the name of the test, an optional description (which is only displayed if present in the instance), whether all tests have passed, for quick inspection, and the list of each test with an indication if it has not passed:

```
<group>
    <label class="title" ref="@name"/>
    <output class="block" ref="description"/>
    <output class="{@pass}" ref="@pass"/>
    <repeat ref="test">
        <output value="."/> → <output ref="@res"/>
        <output class="wrong"
                value="if(@pass!='yes', concat(' expected: ', ▶
@req), '')"/>
    </repeat>
</group>
```

This looks like this when run:

Success	Failure
	

compare() function

PASS

```
compare(apple, orange) → -1
compare(orange, apple) → 1
compare(apple, apple) → 0
compare(apple, ) → 1
compare(, apple) → -1
compare(, ) → 0
compare(1, 2) → -1
compare(2, 1) → 1
compare(2, 10) → 1
compare(10, 2) → -1
compare(1, -1) → 1
compare(apple, APPLE) → 1
compare(cafe, café) → -1
compare(small, 大) → -1
compare(大, small) → 1
compare(大, 大) → 0
compare(大, 小) → -1
compare(小, 大) → 1
```

seconds-from-dateTime() function

FAIL

```
1970-01-01T00:00:00Z → 0
1970-01-01T00:00:00 → 0
1969-12-31T23:59:59 → -1
1970-01-01T00:00:00-08:00 → -2880000 expected: 28800
1970-01-01T00:01:00+00:01 → 120 expected: 0
1970-01-01T00:00:00-00:01 → -60
1970-01-01T00:00:00-00:02 → -120
1970-01-01T00:01:00-00:02 → -60
1970-01-01T00:02:00-00:02 → 0
1970-01-01T00:01:00-00:01 → 0
```

4.2. Analysis

Not all test cases can be structured like this, but many can, and even tests that do not test functions can emulate this behaviour by writing results to the tests instance, and use the same mechanism for checking. These forms are introspective: they are requiring the XForms processor to reveal properties of itself, to itself. Although there is no *direct* input from the user, the input in this case can be seen as being the XForms processor itself.

5. Conclusion

In the introduction, the comparison of XForms's separation of data and controls with the separation of style and content using style sheets was not accidental. Someone typing a publishing contract into a word-processor may only be interested in the content; a designer styling the contract may only be interested in how it look; someone in the publishing industry may only be interested in the name of the author, and the sizes of the advance and the royalty being granted: a document may have several layers of abstraction. It is data-driven forms that can represent one of those layers. The dynamism afforded by computer-based forms has blurred the distinction between form and application, and allowed similar techniques and mechanisms to be applied to both.

As shown in the three cases here, the use of presence and relevance of data elements to drive the controls of the interface with the user gives a lot of power, and affords the declarative definition of applications that would normally be thought of as procedural in nature.

6. References

Bibliography

[1] *CSS Snapshot 2017*. Tab Atkins Jr. et al.. W3C. 2017. https://www.w3.org/TR/css-2017/ .

[2] *Forms that Work*. Caroline Jarrett and Gerry Gaffney. Morgan Kaufmann. 2009.

[3] *XML Path Language*. Anders Berglund et al.. W3C. 2010. https://www.w3.org/TR/xpath20/ .

[4] *XForms 1.0*. Micah Dubinko et al.. W3C. 2003. https://www.w3.org/TR/2003/REC-xforms-20031014/ .

[5] *XForms 1.1*. John M. Boyer et al.. W3C. 2009. http://www.w3.org/TR/2009/REC-xforms-20091020/ .

[6] *XForms 2.0*. Erik Bruchez et al.. W3C. 2017. https://www.w3.org/community/xformsusers/wiki/XForms_2.0.

[7] *XForms 1.0 Test Suite*. W3C. 2003. https://www.w3.org/MarkUp/Forms/Test/XForms1.0/Edition3/front_html/XF103edTestSuite.html.

[8] *XForms 1.1 Test Suite*. W3C. 2009. https://www.w3.org/MarkUp/Forms/Test/XForms1.1/Edition1/driverPages/html/.

[9] *reveal*. http://lab.hakim.se/reveal-js .

[10] *remark*. https://remarkjs.com/ .

[11] *webslides*. https://webslides.tv/ .

[12] *deck*. http://imakewebthings.com/deck.js .

[13] *shwr*. https://shwr.me/ .

tokenized-to-tree

An XProc/XSLT Library For Patching Back Tokenization/Analysis Results Into Marked-up Text

Gerrit Imsieke
le-tex publishing services GmbH
<gerrit.imsieke@le-tex.de>

Abstract

This paper presents an XProc/XSLT library for performing string-based tokenization and analysis (TA) on marked-up text, represented as XML, and possibly deeply nested. Tokenization and analysis yield another XML representation of the input that overlaps with the original markup. These results need to be merged with the original markup. The task is made complicated because the input needs to be normalized prior to TA, for example by converting non-breaking and other typographic spaces to plain spaces, ignoring index entries, or processing footnotes separately. After the TA results have been merged into the normalized source XML, things that have been normalized away need to be restored.

The library provides a representation for the normalized input units (typically paragraphs) and for different types of placeholders. Three applications that build on this representation are presented: Linguistic TA of OOXML (MS Word) files, inserting line numbers from a PDF rendering into its TEI source, and linking occurrences of headwords in reference work entries to their primary entries.

The process of normalizing, character position counting, TA invocation, patching back the TA results, and inverting the normalization, is complex. It consists of multiple XSLT passes that need to be customized and assembled in a distinct way for each application. Encapsulating invariant core process steps as well as macroscopic, customizable steps and orchestrating the XML transformation steps in a sometimes non-linear way requires a technology that is good at these things. In this regard, the paper, and the open-source library that the presented applications are built on, is a demonstration of the utility of XProc in complex publishing pipelines.

Keywords: Overlapping Markup, Automatic Linking, TEI, XProc, XSLT, Regular Expression, Pagination, Publishing, Whitespace Normalization, Tokenization, XML Splitting, Mixed Content

1. Introduction

Tokenization/analysis (TA) tasks can be be performed with relative ease on XML elements that contain only text nodes. Examples include:

- Using a list of headwords in a reference work, with corresponding link targets to the entries, in order to find these words in other entries and link them to the main entry;
- Running a text through natural language analysis that returns a sequence of tokens along with part-of-speech information for each token, adding this TA information to the source text;
- Splitting a PDF rendering of a text-critical edition into lines and identifying which string positions in the source text correspond to the beginning of each PDF line; adding line break markers to the TEI source.

An established tool, at least for XSLT 2.0+ users, would be `xsl:analyze-string`. The search terms or PDF lines will be converted into regular expressions, and the matching strings will be tagged appropriately. There are also other tools than XSLT/XPath/XQuery-based regular expression matching. Natural language processing, as performed by Apache NLP [1], will do the tokenization and the tagging of an input string all by itself.

There is a significant shortcoming with these tools though when applied to mixed-content XML: They will not honor existing markup. They will rather process the input as a flat string and add only statically configured markup in the tagged replacement text.

Example 1. Matching reference work headwords and linking to their entries

Suppose there is a pharmaceutical reference work with entries for solutions of varying concentrations. Another entry or publication contains this sentence:

```
<p>375 ml of a sodium chloride solution (0.455 mol · l<sup>-1</sup>)
is diluted with 0.5 l of water</p>
```

The list of headwords / link targets contains this entry:

```
<link linkend="preparation-00437">sodium chloride solution
(0.455 mol · l<sup>-1</sup>)</link>
```

Accounting for differences in whitespace that may occur in the text, the link tagging can be performed using the following regular expression that is generated from the search term:

```
sodium[\s\p{Zs}]+chloride[\s\p{Zs}]+solution[\s\p{Zs}]+\(0.455[\s\p{Zs}]
+mol[\s\p{Zs}]*·[\s\p{Zs}]*l-1\)
```

Tagging will be lost using this method. Even if the search term's tagging will be used for reconstructing the match in the output, there will be (at least) two issues:

- Using `xsl:analyze-string` or other regex-matching methods, the non-matching parts of the input will just be rendered as text nodes.
- Subtle tagging variations within the matches will not be accounted for in the output.

As an example for the second issue, consider this matching source markup that contains a processing instruction for typesetting:

```
sodium chloride solution<?tex \break ?> (0.455 mol · l<sup>-1</sup>)
```

If the line break that this processing instruction enforces were missing after automatic link insertion, the pagination might change which would amount to a major issue for this class of reference works that sometimes comprise thousands of pages.

2. Requirements

The main requirements for a non-destructive tagging are:

- existing markup must be preserved
  - within and outside the matching tokens,
  - no matter how deep the nesting;
- existing text nodes must be preserved; in particular, non-breaking or thin spaces must not be replaced with plain spaces that the search term may contain;
- if the newly introduced markup overlaps existing markup, either the new or the old markup needs to be split into segments. This happens frequently when re-inserting linguistic analysis results to markup.

Example 2. Linguistic Analysis

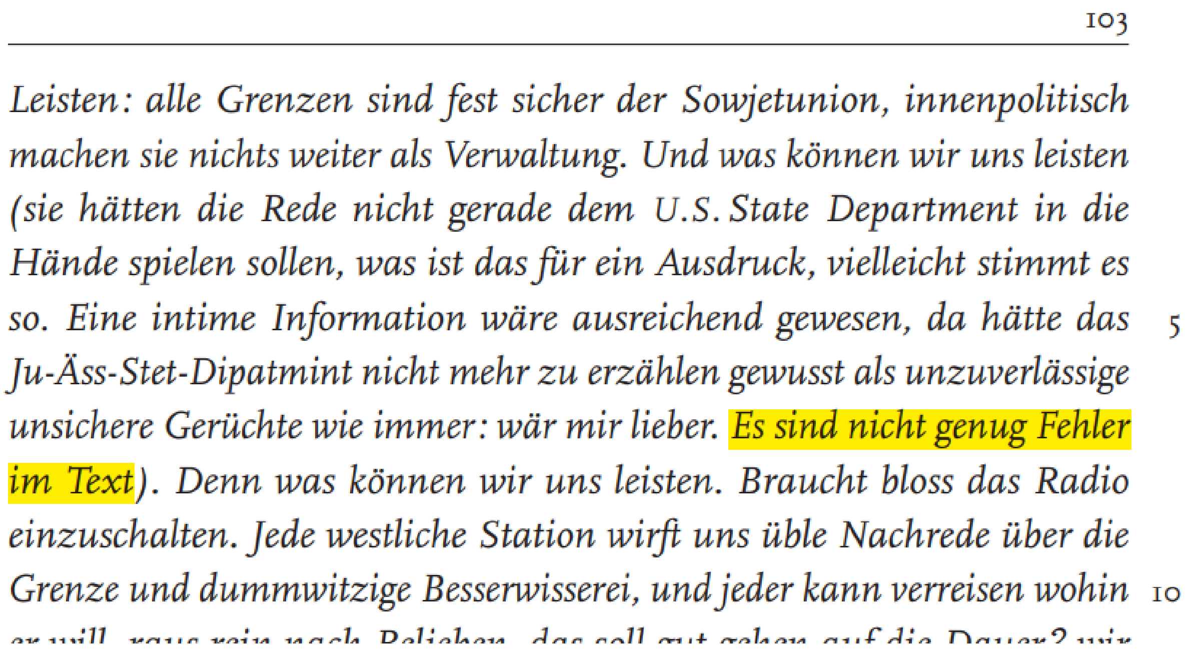
A contrived example for the latter processing:

```
<p>partly <b>bold te</b>xt</p>
```

After linguistic analysis, it becomes:

```
<p><word pos="adv">partly</word> <word pos="adj"><b>bold</b></word><b>
 </b><word pos="noun"><b>te</b>xt</word></p>
```

In many XML vocabularies, content is included in the linear markup stream at a position that it is anchored to, while it is supposed to be rendered elsewhere. Think of index terms or footnotes. When linking, doing linguistic analysis, or patching the PDF line breaks into the XML document, this kind of content must be ignored (apart from the footnote marker, in the line break scenario). It must be re-inserted after the tokenization and analysis results have been applied.



103

Leisten: alle Grenzen sind fest sicher der Sowjetunion, innenpolitisch machen sie nichts weiter als Verwaltung. Und was können wir uns leisten (sie hätten die Rede nicht gerade dem U.S. State Department in die Hände spielen sollen, was ist das für ein Ausdruck, vielleicht stimmt es so. Eine intime Information wäre ausreichend gewesen, da hätte das 5
Ju-Äss-Stet-Dipatmint nicht mehr zu erzählen gewusst als unzuverlässige unsichere Gerüchte wie immer: wär mir lieber. Es sind nicht genug Fehler im Text). Denn was können wir uns leisten. Braucht bloss das Radio einzuschalten. Jede westliche Station wirft uns üble Nachrede über die Grenze und dummwitzige Besserwisserei, und jeder kann verreisen wohin 10
er will, raus rein nach Belieben, das soll gut gehen auf die Dauer? wir

Taken from: *Uwe Johnson Werkausgabe. Ein Vorhaben der Berlin-Brandenburgischen Akademie der Wissenschaften an der Universität Rostock*

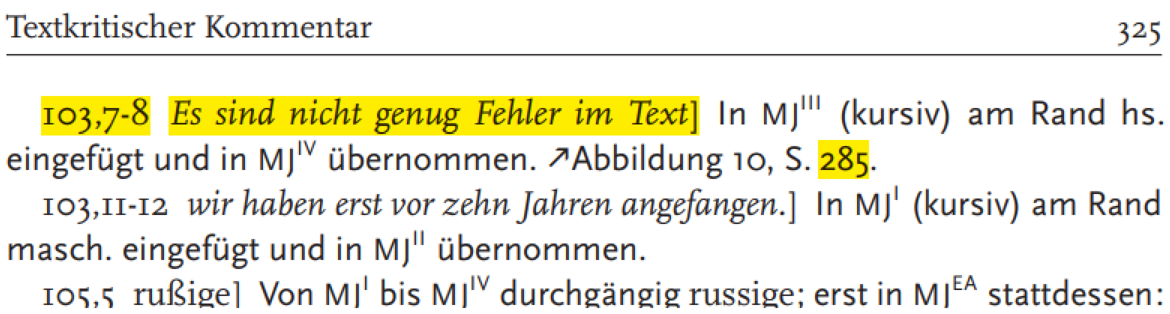
Figure 1. The rendered source text that the text-critical note refers to

2.1. A detailed example: Marking up line numbers in PDF; matching generated text

An inverse problem is that of generated text. It affects the PDF line break problem in particular. The footnote markers mentioned above are often generated text. The actual footnote number may be calculated by applying the same rules that the typesetting system used for the PDF rendering. Inserting this generated text in a modified source XML allows a matching between the source XML and the PDF rendering. However, this exact match is not possible for page and line numbers that serve as a reference to the original text (see Figure 1) in a text-critical note, as depicted in Figure 2.

The problem here is that the customer not only wants the line numbers patched back to the TEI source for the main text, but also for the text-critical notes that are rendered at a different page range in the PDF.

The typesetting system used to generate the layout depicted above, which is LaTeX, is able to determine page and line numbers of any given rendered text. In principle, one could use the auxiliary file in which LaTeX stores the numbers for each referred item in order to adorn this item in the source XML with its exact position, thus allowing an exact matching between source XML and PDF output. We chose not to do this, but rather detect these types of references in the PDF and replace them with generic regular expressions such as `\[\d+,\d+(\p{Pd}\d+)?\]` instead of a regular expression that would have only matched the exact numbers



Textkritischer Kommentar 325

103,7-8 *Es sind nicht genug Fehler im Text*] In MJ^{III} (kursiv) am Rand hs. eingefügt und in MJ^{IV} übernommen. ↗Abbildung 10, S. 285.

103,11-12 *wir haben erst vor zehn Jahren angefangen.*] In MJ^{I} (kursiv) am Rand masch. eingefügt und in MJ^{II} übernommen.

105,5 rußige] Von MJ^{I} bis MJ^{IV} durchgängig russige; erst in MJ^{EA} stattdessen:

Taken from: *Uwe Johnson Werkausgabe. Ein Vorhaben der Berlin-Brandenburgischen Akademie der Wissenschaften an der Universität Rostock*

Figure 2. A text-critical note referring to page 103, lines 7–8, and to a figure on page 285

in the source XML. Instead of prepending the exact numbers to the note in the source XML, we simply prepend '1,1' to the note's XML content and flag it as generated text so that it can be removed again in a later processing stage.

3. A detailed look at the PDF line number patching example

At this point it may be instructive to look at how this particular process, feeding back line breaks into the source XML, is orchestrated.

Most of the configuration happens by custom XSLT that imports some of the library's XSLT stylesheets. For example, the XSLT on the `prepare-input-xsl` port imports a library XSLT that is handles TEI, which in turn imports the basic prepare-input stylesheet.

There may be separate configurations for different portions of the text. The pages to be processed and the interesting regions within a page (excluding running heads and page numbers) can be configured in the `postprocess-poppler-params` XML file (which is an XProc `p:param-set`). Consequently, the whole book will be processed in separate invocations of the `ttt:mark-linebreaks` pipeline. Of course these separate runs with their distinct settings are also assembled as an XProc pipeline.

As an example for different configurations for different parts of the boook, we look at the `ttt:prepare-input` step. It has three distinct substeps, corresponding to different XSLT modes:

1. `ttt:add-ids`: Every element (plus comments and processing instructions) need to get an ID for later merging);
2. `ttt:discard`: Only interesting elements will be retained. For the main text of the novel, it is `<p>` and `<head>`, for the critical apparatus, it is `<note type="printnote" subtype="TextKritKomm">`. These variations in configuration will be accounted for by importing a more generic XSLT and adding/

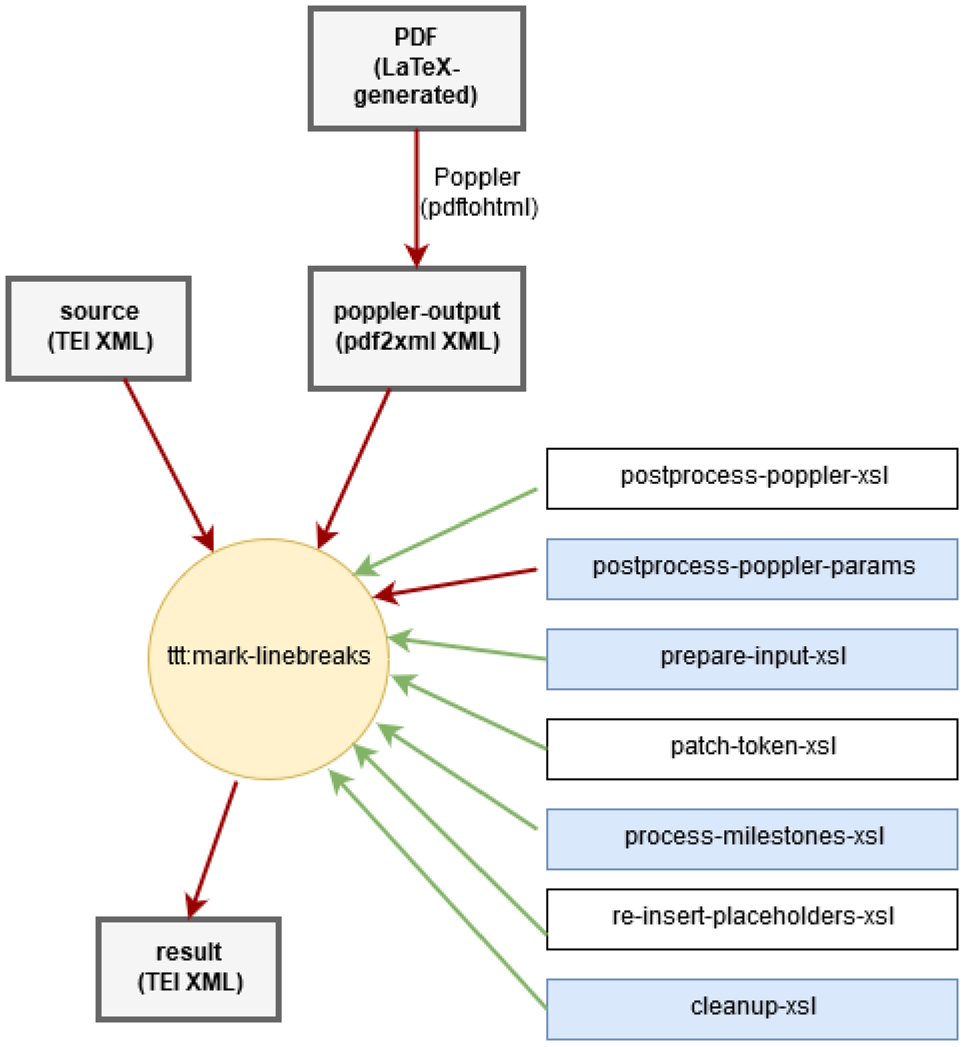


Figure 3. Interface of the `ttt:mark-linebreaks` XProc step (see next figure for a key)

overwriting some templates. in this mode, also special operations such as whitespace normalization and inserting generated text will take place;

3. `ttt:count-text`: The extracted and normalized text will be counted. This is a crucial operation that adds `ttt:start` and `ttt:end` attributes by which tokenization and analysis results can be re-attached to the nodes that they correspond to. Remember that tokenization and analysis will only be performed on strings, therefore we need the string value of each paragraph-like processing unit (PU) alongside the length-annotated, normalized, tagged input for this paragraph.

The detailed structure within `ttt:mark-linebreaks` looks like on Figure 4.

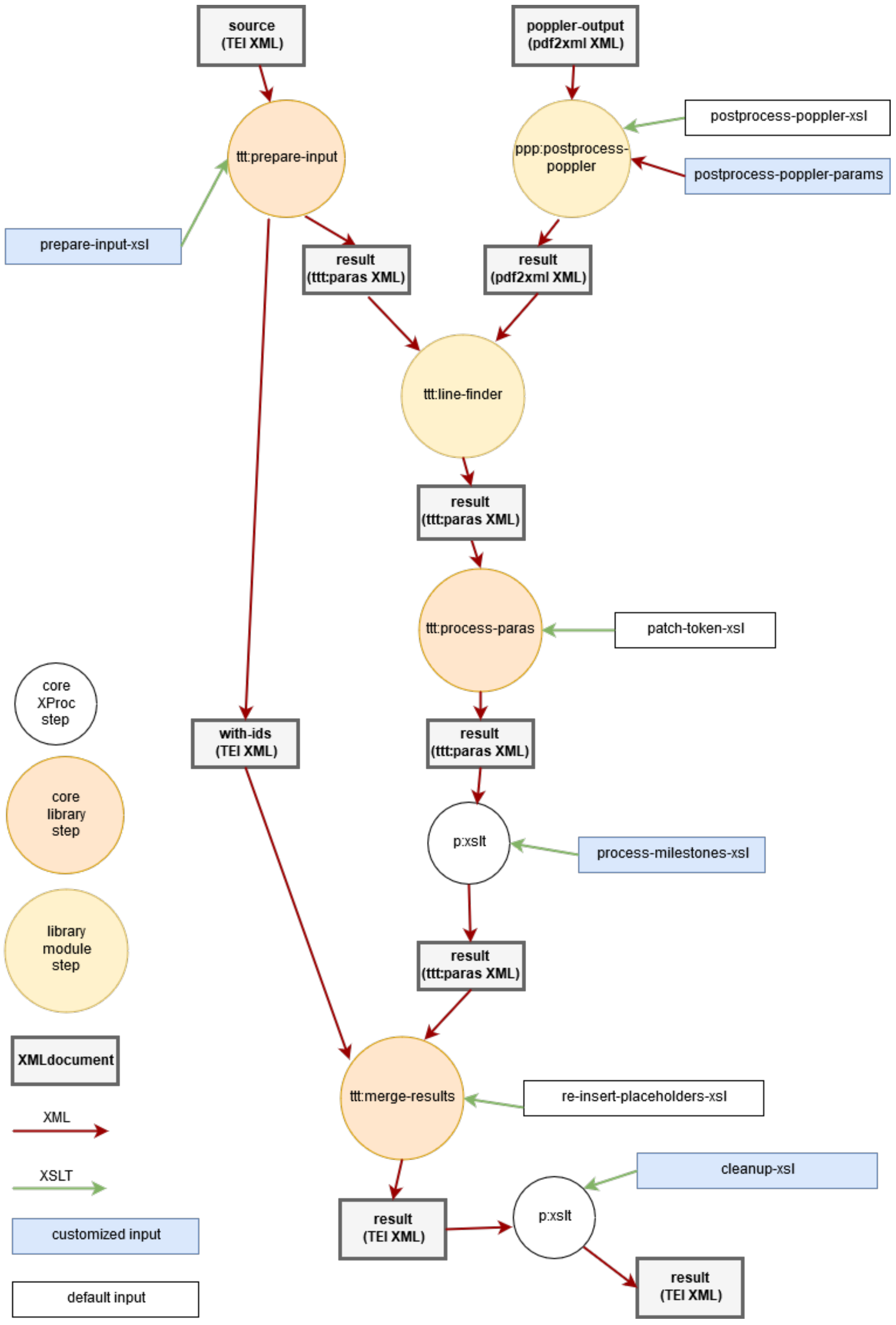
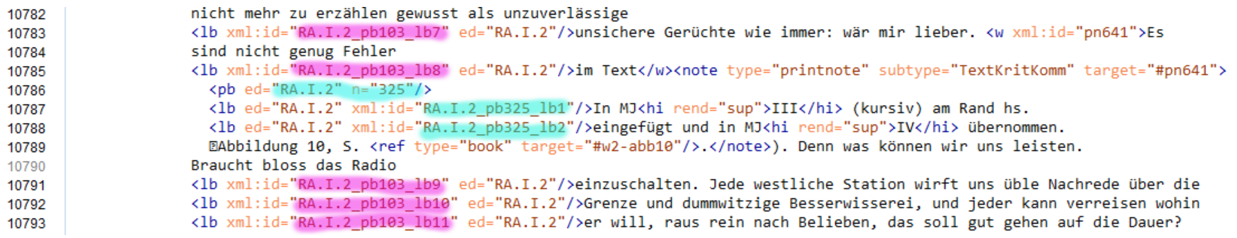


Figure 4. Detailed structure of the `ttt:mark-linebreaks` XProc step

This length-annotated markup mentioned in the list item about `ttt:count-text` will be processed in the `ttt:process-paras` step. Its preceding step, in this example `ttt:line-finder`, provides or invokes the task-specific tokenization/analysis routine (Apache NLP or some custom XSLT that marks link candidates, respectively, for the two other use cases discussed in this paper) and augments the intermediate `<ttt:paras>` document that will be dicussed in greater detail in Section 4.

Just to give an illustration of its effect on the TEI source document:



```
        nicht mehr zu erzählen gewusst als unzuverlässige
        <lb xml:id="RA.I.2_pb103_lb7" ed="RA.I.2"/>unsichere Gerüchte wie immer: wär mir lieber. <w xml:id="pn641">Es
        sind nicht genug Fehler
        <lb xml:id="RA.I.2_pb103_lb8" ed="RA.I.2"/>im Text</w><note type="printnote" subtype="TextKritKomm" target="#pn641">
          <pb ed="RA.I.2" n="325"/>
          <lb ed="RA.I.2" xml:id="RA.I.2_pb325_lb1"/>In MJ<hi rend="sup">III</hi> (kursiv) am Rand hs.
          <lb ed="RA.I.2" xml:id="RA.I.2_pb325_lb2"/>eingefügt und in MJ<hi rend="sup">IV</hi> übernommen.
          Abbildung 10, S. <ref type="book" target="#w2-abb10"/>.</note>). Denn was können wir uns leisten.
        Braucht bloss das Radio
        <lb xml:id="RA.I.2_pb103_lb9" ed="RA.I.2"/>einzuschalten. Jede westliche Station wirft uns üble Nachrede über die
        <lb xml:id="RA.I.2_pb103_lb10" ed="RA.I.2"/>Grenze und dummwitzige Besserwisserei, und jeder kann verreisen wohin
        <lb xml:id="RA.I.2_pb103_lb11" ed="RA.I.2"/>er will, raus rein nach Belieben, das soll gut gehen auf die Dauer?
```

Taken from: *Uwe Johnson Werkausgabe. Ein Vorhaben der Berlin-Brandenburgischen Akademie der Wissenschaften an der Universität Rostock*

Figure 5. The effect of the two `ttt:mark-linebreaks` passes (on the main text and on the notes). In a color rendering of this paper, line breaks in notes are cyan (two in the center) while line breaks in the main text are purple (two on the left-hand side on top; three below the note. A page break in the note has also been captured by the process

4. The tokenized-to-tree library

The library consists of three macroscopic XProc steps. Each project needs to add a bespoke TA step to run after the first, normalization step, and there may be post-processing or cleanup XSLT passes after the second and third library steps. The steps are:

4.1. Step `ttt:prepare-input`

Transforms the content to a representation with wrappers around each processing unit (read: paragraph) and with placeholders for footnotes, processing instructions and the like, and with single plain space placeholders for multiple or typographic spaces.

Unless already present, IDs will be added to all elements that may be turned into placeholders. This is because all PUs, for example a paragraph with a footnote and the paragraph *in* the footnote, will be in `ttt:para` that form a flat list after normalization. The normalized processing untits contain placeholders with IDs, and these IDs allow reconstruction of the original document after all other operations have completed.

The normalizing XSLT is provided to the step on an input port. The libarary currently contains basic normalization stylesheets for DocBook and TEI. They can be adapted to a project's needs by importing them, or they can be replaced with XSLT code that supports different XML vocabularies.

An important aspect of the normalization is the post-normalization string position counting that adds start and end position numbers to each element, as attributes in the `ttt` namespace.

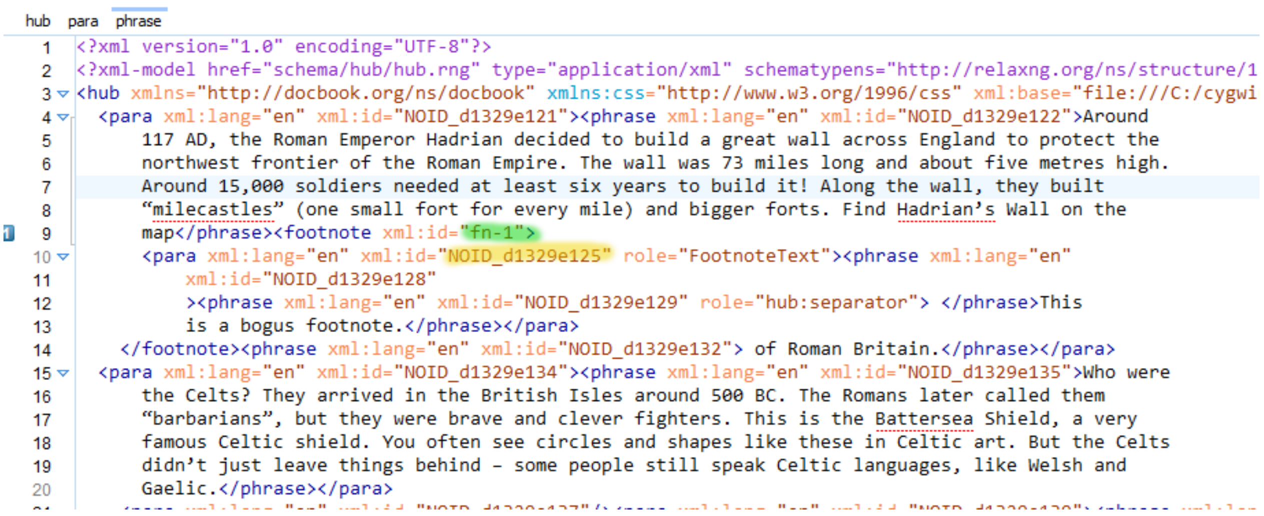
More details about the three XSLT passes (ID generation, deletion of uninteresting content, text counting) within this step can be found in the previous section, Section 3.

This step has two output ports: The port `result` will provide the `ttt:para` list with the normalized PUs, wrapped in a `ttt:paras` top-level element, and the port `with-ids` will provide the ID-enriched original document that is needed for the final placeholder expansion.

A note on the particular suitability of XProc

Apart from the encapsulation that XProc offers, and that any other functional language will offer, XProc has an innate concept of multiple function return values (output ports). The different return values (= documents on output ports) need not be multiplexed or consumed simultaneously. This is a huge benefit for processes where a secondary output, in this example from the `with-ids` port, will be used by another step at a later stage (see Section 4.5).

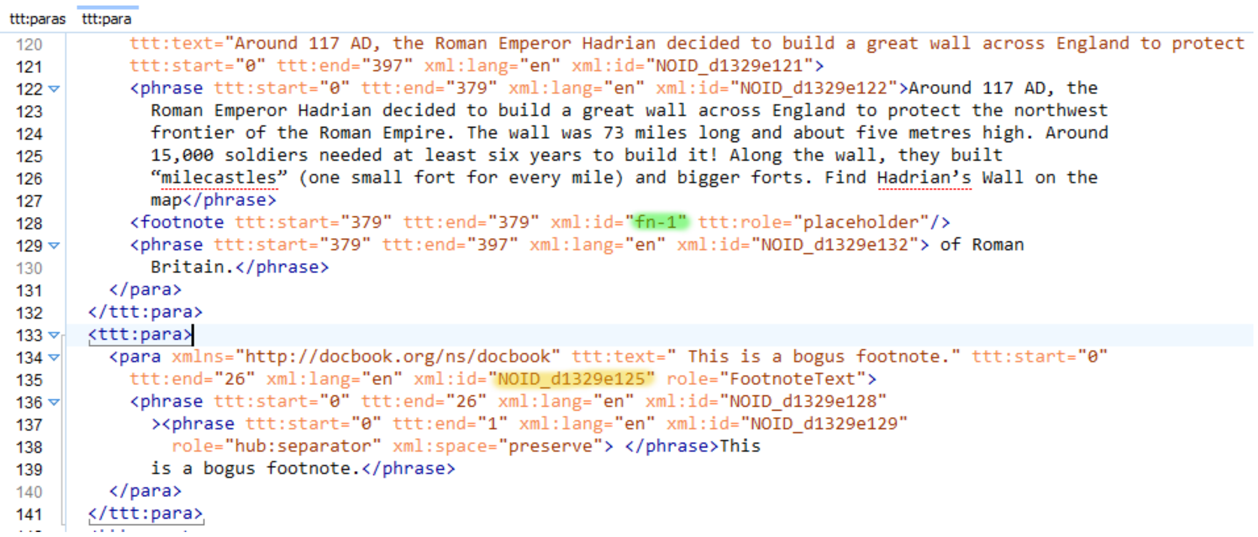
For the linguistic analysis example, parts of the `with-ids` output are depicted in Figure 6. (See Section 5 for how this looks in Word.)



```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="schema/hub/hub.rng" type="application/xml" schematypens="http://relaxng.org/ns/structure/1
<hub xmlns="http://docbook.org/ns/docbook" xmlns:css="http://www.w3.org/1996/css" xml:base="file:///C:/cygwi
  <para xml:lang="en" xml:id="NOID_d1329e121"><phrase xml:lang="en" xml:id="NOID_d1329e122">Around
      117 AD, the Roman Emperor Hadrian decided to build a great wall across England to protect the
      northwest frontier of the Roman Empire. The wall was 73 miles long and about five metres high.
      Around 15,000 soldiers needed at least six years to build it! Along the wall, they built
      “milecastles” (one small fort for every mile) and bigger forts. Find Hadrian’s Wall on the
      map</phrase><footnote xml:id="fn-1">
      <para xml:lang="en" xml:id="NOID_d1329e125" role="FootnoteText"><phrase xml:lang="en"
          xml:id="NOID_d1329e128"
          ><phrase xml:lang="en" xml:id="NOID_d1329e129" role="hub:separator"> </phrase>This
          is a bogus footnote.</phrase></para>
    </footnote><phrase xml:lang="en" xml:id="NOID_d1329e132"> of Roman Britain.</phrase></para>
  <para xml:lang="en" xml:id="NOID_d1329e134"><phrase xml:lang="en" xml:id="NOID_d1329e135">Who were
      the Celts? They arrived in the British Isles around 500 BC. The Romans later called them
      “barbarians”, but they were brave and clever fighters. This is the Battersea Shield, a very
      famous Celtic shield. You often see circles and shapes like these in Celtic art. But the Celts
      didn’t just leave things behind – some people still speak Celtic languages, like Welsh and
      Gaelic.</phrase></para>
```

Figure 6. The (partly indented) output on the `with-ids` port of the `ttt:prepare-input` step for the natural language processing example

The normalized input (after the `ttt:count-text` XSLT pass) for this example looks like this:



```
    ttt:text="Around 117 AD, the Roman Emperor Hadrian decided to build a great wall across England to protect
    ttt:start="0" ttt:end="397" xml:lang="en" xml:id="NOID_d1329e121">
    <phrase ttt:start="0" ttt:end="379" xml:lang="en" xml:id="NOID_d1329e122">Around 117 AD, the
      Roman Emperor Hadrian decided to build a great wall across England to protect the northwest
      frontier of the Roman Empire. The wall was 73 miles long and about five metres high. Around
      15,000 soldiers needed at least six years to build it! Along the wall, they built
      “milecastles” (one small fort for every mile) and bigger forts. Find Hadrian’s Wall on the
      map</phrase>
    <footnote ttt:start="379" ttt:end="379" xml:id="fn-1" ttt:role="placeholder"/>
    <phrase ttt:start="379" ttt:end="397" xml:lang="en" xml:id="NOID_d1329e132"> of Roman
      Britain.</phrase>
  </para>
</ttt:para>
<ttt:para>
  <para xmlns="http://docbook.org/ns/docbook" ttt:text=" This is a bogus footnote." ttt:start="0"
    ttt:end="26" xml:lang="en" xml:id="NOID_d1329e125" role="FootnoteText">
    <phrase ttt:start="0" ttt:end="26" xml:lang="en" xml:id="NOID_d1329e128"
      ><phrase ttt:start="0" ttt:end="1" xml:lang="en" xml:id="NOID_d1329e129"
        role="hub:separator" xml:space="preserve"> </phrase>This
      is a bogus footnote.</phrase>
  </para>
</ttt:para>
```

Please note the `footnote` element that is merely a placeholder in the paragraph that it appears in. The paragraph that the footnote contained is in a separate `ttt:para` child of the `ttt:paras` element.

> **Why are there separate runs for inserting the TEI line breaks?**
> In contrast to the TEI example in Section 3, main text and footnotes are processed in a single run for linguistic analysis. The two-run invocation of the line number scenario is necessitated by two factors:
>
> - the need for processing contiguous chunks of the PDF with slightly different settings for the text-critical notes;
> - most importantly, because the algorithm does not iterate over the PDF lines and tries to find them in the yet-unmarked XML source. It rather finds matches in all processing unit strings for a given regular expression derived from a PDF line. Then it filters away implausible match candidates in order to achieve an optimal coverage of the normalized (stringified) input units. Plausibilization is done by looking whether subsequent PDF lines correspond to subsequent matches. Therefore it is important to process the endnote-like text-critical notes, both their XML source elements and their PDF page range, in a separate pass.

The question why there is DocBook XML to be seen when the task is to linguistically analyze Word files will be answered in Section 5.

4.2. (project-specific TA step)

This step will put each processing unit’s TA result in parallel to the normalized input, as the second child in the `ttt:para` element that already wraps the normalized processing unit.

The step must make sure that the TA results will be represented in a specific XML vocabulary: `ttt:tokens` elements with untagged text and `ttt:t` elements for the individual tokens. The `ttt:t` elements are expected to have `@start` and `@end` attributes for the string positions, in order to facilitate the merging of both XML structures. Other attribtues on `ttt:t` contain the analysis results such as part-of-speech, corresponding PDF line/page numbers, or link targets.

4.3. Step `ttt:process-paras`

The name of the XProc file that this step is declared in is probably a bit more telling: `ttt-3-integrate-tokenizer-results.xpl`. For the time being, the step will retain its historically grown name.

Naming things unmistakably is difficult anyway in this library, in particular when it comes to merging. There are two merging operations: The first (this step) combines the normalized input with the TA results. The other (the final step) restores the original document structure by re-filling the placeholders.

The first thing that `ttt:process-paras` does is to validate the project-specific output against the Relax NG schema[1] for the `ttt:paras` side-by-side document. This will help people get their project-specific TA output right.

Then the step applies two to three XSLT passes to the content, typically using the default stylesheet that comes with the step:

Mode `ttt:patch-token-results`

This pass will insert `ttt:start`/`ttt:end` milestone elements into the `ttt:para/*[1]` branch that contains the normalized processing units. The string positions where each milestone element will appear corresponds to the `ttt:t/@start`/`ttt:t/@end` attribute values for each token in the `ttt:para/ttt:tokens` branch. The `ttt:para/ttt:tokens` becomes dispensable after this pass and will be removed. The surrounding `ttt:para` element is not needed any more and may also be unwrapped.

Although the concepts and the arithmetic of this token-merging step is primary-school level, it is nevertheless good to encapsulate it in an XProc step, together with the normalization format validation. If it were carried out on an ad-hoc basis for each project, chances are that off-by-one or more severe errors will creep in.

[1] https://github.com/transpect/tokenized-to-tree/blob/master/schema/tokenized-to-tree.rng

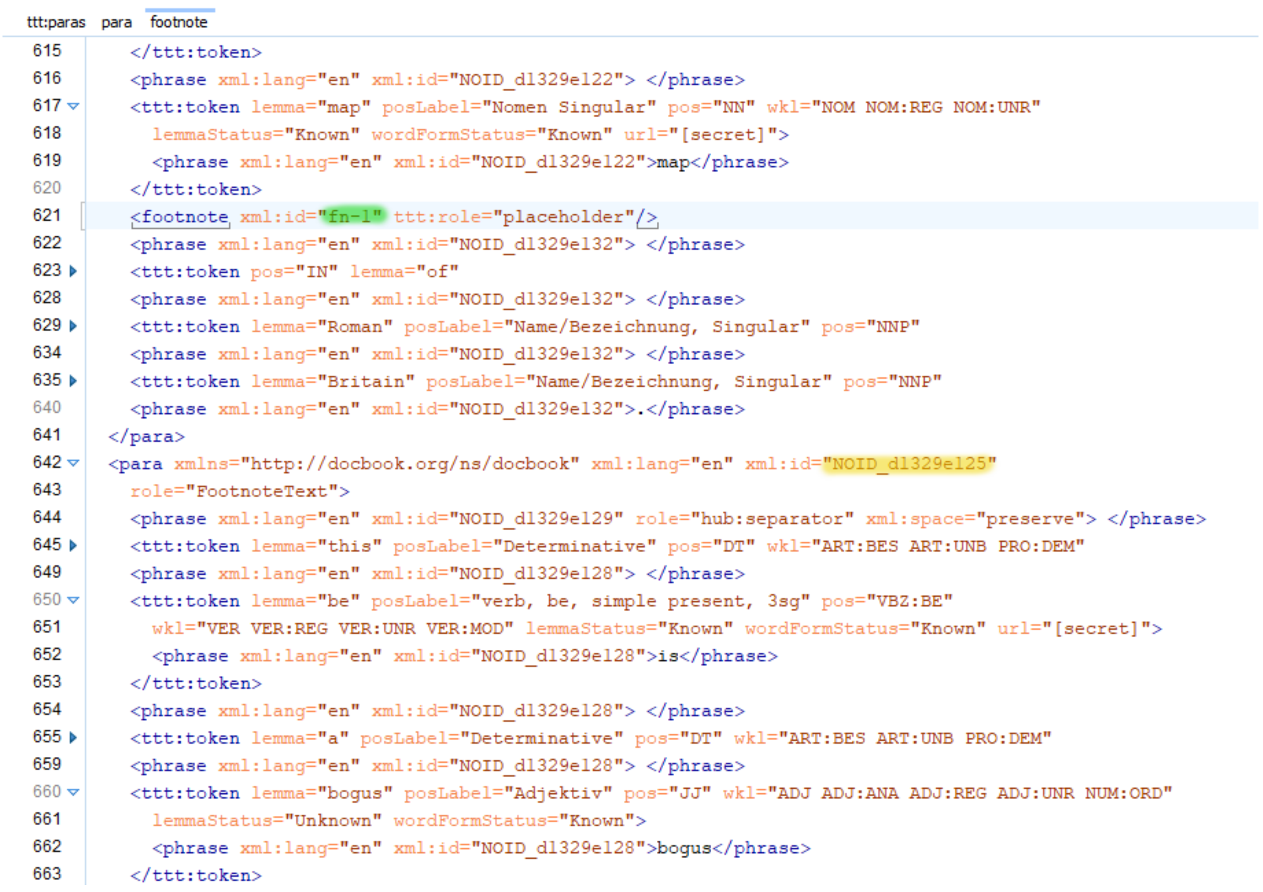
Mode `ttt:eliminate-duplicate-start-end-elts`	This is just a technical cleanup mode for removing duplicate start or end milestones.
Mode `ttt:pull-up-delims`	This is an optional `p:xslt` step that can be deactivated by setting the step option `milestones-only="yes"`. It is needed for the linguistic analysis and for the linking scenarios, but not for the line number insertion scenario. In this XSLT pass, the milestone elements that have been inserted into text nodes at any depth of the processing units will be pulled up so that they reside immediately beneath their PU's top-level element, splitting any intermediate elements. (If they should not be pulled up this far for a specific project, the project needs to override a template.) This technique has been called "upward projection" by the author. An example can be found at [2]. Other than in the xsl-list example, the milestones will not simply be pulled up, but they will form a new `ttt:token` element that inherits the milestones' attribute and that engulfs the (potentially deeply nested) markup that the pull-up operation has split.

A note on input segmentation

The performance of upward projection scales roughly with the product of input length (node count) and the number of splitting points. (The proof of this conjecture is left to another publication.)

From a performance standpoint, it is therefore advisable to split the input into smaller processing units, for example on a paragraph level. Suppose that the input consists of ten paragraphs, each of them comprising 20 nodes and four milestones to be pulled up. If one were to process the input as a whole, the time needed would be about (10×20)×(10×4) = 8000 arbitrary units, while when processing them individually, the time needed is only 10×20×4 = 800 a.u.

An indented output of this step for the NLP example can be seen in Figure 7.



```
    </ttt:token>
    <phrase xml:lang="en" xml:id="NOID_d1329e122"> </phrase>
    <ttt:token lemma="map" posLabel="Nomen Singular" pos="NN" wkl="NOM NOM:REG NOM:UNR"
      lemmaStatus="Known" wordFormStatus="Known" url="[secret]">
      <phrase xml:lang="en" xml:id="NOID_d1329e122">map</phrase>
    </ttt:token>
    <footnote xml:id="fn-1" ttt:role="placeholder"/>
    <phrase xml:lang="en" xml:id="NOID_d1329e132"> </phrase>
    <ttt:token pos="IN" lemma="of"
    <phrase xml:lang="en" xml:id="NOID_d1329e132"> </phrase>
    <ttt:token lemma="Roman" posLabel="Name/Bezeichnung, Singular" pos="NNP"
    <phrase xml:lang="en" xml:id="NOID_d1329e132"> </phrase>
    <ttt:token lemma="Britain" posLabel="Name/Bezeichnung, Singular" pos="NNP"
    <phrase xml:lang="en" xml:id="NOID_d1329e132">.</phrase>
  </para>
  <para xmlns="http://docbook.org/ns/docbook" xml:lang="en" xml:id="NOID_d1329e125"
    role="FootnoteText">
    <phrase xml:lang="en" xml:id="NOID_d1329e129" role="hub:separator" xml:space="preserve"> </phrase>
    <ttt:token lemma="this" posLabel="Determinative" pos="DT" wkl="ART:BES ART:UNB PRO:DEM"
    <phrase xml:lang="en" xml:id="NOID_d1329e128"> </phrase>
    <ttt:token lemma="be" posLabel="verb, be, simple present, 3sg" pos="VBZ:BE"
      wkl="VER VER:REG VER:UNR VER:MOD" lemmaStatus="Known" wordFormStatus="Known" url="[secret]">
      <phrase xml:lang="en" xml:id="NOID_d1329e128">is</phrase>
    </ttt:token>
    <phrase xml:lang="en" xml:id="NOID_d1329e128"> </phrase>
    <ttt:token lemma="a" posLabel="Determinative" pos="DT" wkl="ART:BES ART:UNB PRO:DEM"
    <phrase xml:lang="en" xml:id="NOID_d1329e128"> </phrase>
    <ttt:token lemma="bogus" posLabel="Adjektiv" pos="JJ" wkl="ADJ ADJ:ANA ADJ:REG ADJ:UNR NUM:ORD"
      lemmaStatus="Unknown" wordFormStatus="Known">
      <phrase xml:lang="en" xml:id="NOID_d1329e128">bogus</phrase>
    </ttt:token>
```

Figure 7. The (indented) output of the `ttt:pull-up-delims` step for the natural language processing example

Note that since the document language was set in the input by character-level assignment rather than by Word's paragraph or character styles, the `<phrase xml:lang="en">` ranges that spanned across the paragraphs will be split into small pieces by the upward projection in `ttt:pull-up-delims` mode.

4.4. (project-specific postprocessing step)

While a project might choose to override the XSLT templates in the previous step in order to rename the inserted tokens to elements in the project's vocabulary, it is probably better, from a separation-of-concerns standpoint, to do this in a separate XSLT pass. For very large input files (if memory usage and processing time become an issue), the postprocessing may be performed by overriding templates in the `ttt:pull-up-delims` pass of the `ttt:process-paras` step.

4.5. Step `ttt:merge-results`

This step merges the `with-ids` output from the `ttt:prepare-input` step (see Figure 6) with the previous step's output, using a single XSLT pass.

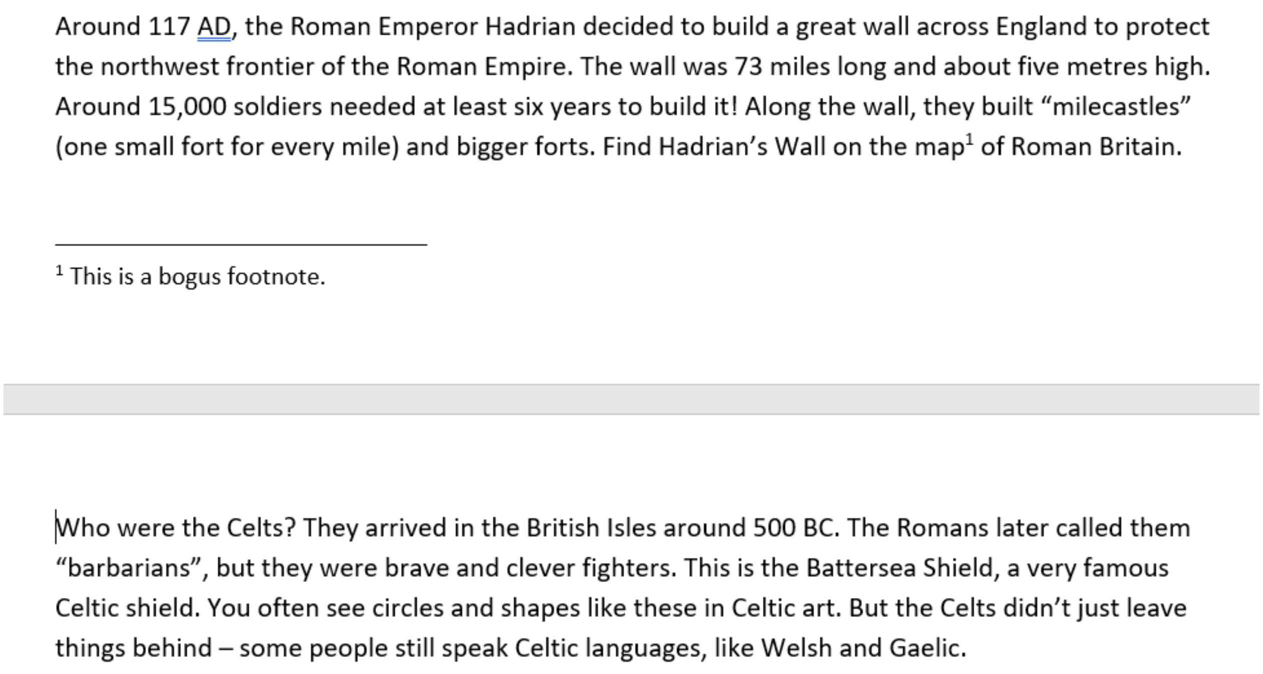
Whenever there is an ID in the `with-ids` document that corresponds to a processing unit's ID in the processed `ttt:paras` document, the processed unit will be included in the output, replacing the original content. The processed units may contain placeholders, for example for footnotes. When such an empty element with an attribute `ttt:role="placeholder"` is encountered, the XSLT pass will use the element's ID to jump back to the `with-ids` document and write the same-id element to the output and process its children. When it encounters an element whose ID matches one of the processed `/ttt:paras/ttt:para/*` elements, it will jump back to the `ttt:paras` document and write the processed paragraph to the output, and so forth.

Taking together Figure 6 and Figure 7, one can try to figure out how this process works. It starts at the top-level `hub` element of the document in Figure 6. When it reaches the `para` that contains the footnote, it replaces it with the same-ID `para` in the `ttt:paras` document. When it encounters the `<footnote ttt:role="placeholder">` element there, it switches back to the `with-ids` document in order to reproduce its `footnote` element. When processing its children, it determines that the contained paragraph is also present in the `ttt:paras` document, and it continues to process this `ttt:paras` paragraph.

5. Details of the linguistic analysis pipeline

The linguistic analysis is used by a German textbook publisher. The XProc pipeline sends the normalized strings to a Web service that invokes Apache NLP [1] for tokenization and analysis and then adorns the identified tokens with information whether a pupil reading the text already knows the lemma and the word form that it appears in (past perfect etc.). This is part of a large vocabulary management platform that BaseX GmbH and LanguageTool's Daniel Naber built for the publisher. The pipeline gets as input a Word manuscript and information about the work that it is part of. It forwards the work identifier to the Web service so that the service is able to look up which words and word forms the pupils of a certain grade in a certain German Bundesland in a certain school form already know.

The input may look like this:



Around 117 AD, the Roman Emperor Hadrian decided to build a great wall across England to protect the northwest frontier of the Roman Empire. The wall was 73 miles long and about five metres high. Around 15,000 soldiers needed at least six years to build it! Along the wall, they built “milecastles” (one small fort for every mile) and bigger forts. Find Hadrian’s Wall on the map[1] of Roman Britain.

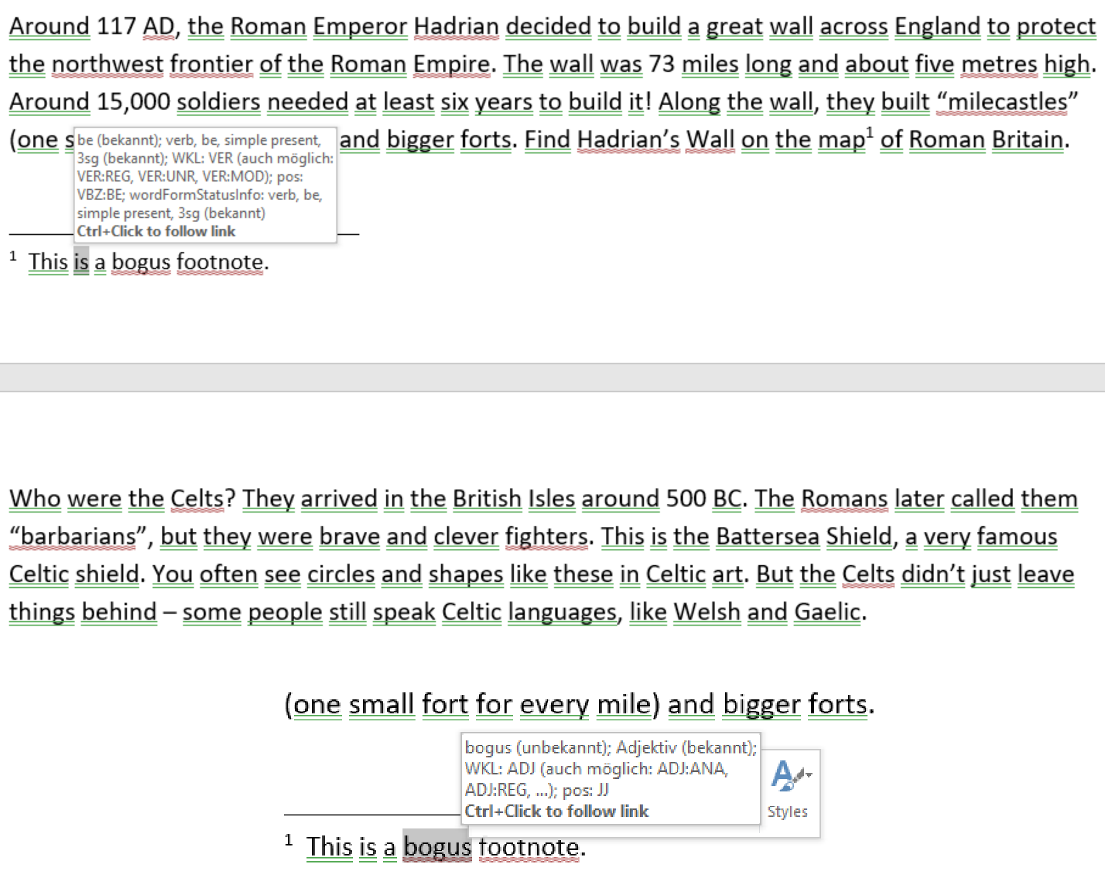
[1] This is a bogus footnote.

Who were the Celts? They arrived in the British Isles around 500 BC. The Romans later called them “barbarians”, but they were brave and clever fighters. This is the Battersea Shield, a very famous Celtic shield. You often see circles and shapes like these in Celtic art. But the Celts didn’t just leave things behind – some people still speak Celtic languages, like Welsh and Gaelic.

The linguistic analysis service adds both linguistic information and the status whether the words and word forms are known to pupils at the given stage. The result can be seen in Figure 7.

The XML format that is sent to the analysis is not OOXML but the DocBook-based Hub format [3]. This is partly because the internal representation and fragmentation of text in OOXML is poorly suited for the analysis, but more importantly because the service is designed to also process InDesign’s IDML format, for which a Hub XML conversion exists.

After analysis has been performed, the user receives a new Word file, created from Hub XML, where the TA results are presented as tooltips on hyperlinks. The links have differently colored double underlines, depending on the analysis results – whether the pupils already know the word or whether the word exists at all in the vocabulary management system. There is a Word add-on (not shown here for reasons of trade secrecy) for editors to inform the system that a word or its form is indeed introduced in the current manuscript. Upon re-upload of the Word file, these status changes will be registered through the vocabulary management Web service, using `p:http-request`.



Around 117 AD, the Roman Emperor Hadrian decided to build a great wall across England to protect the northwest frontier of the Roman Empire. The wall was 73 miles long and about five metres high. Around 15,000 soldiers needed at least six years to build it! Along the wall, they built "milecastles" (one s and bigger forts. Find Hadrian's Wall on the map[1] of Roman Britain.

be (bekannt); verb, be, simple present, 3sg (bekannt); WKL: VER (auch möglich: VER:REG, VER:UNR, VER:MOD); pos: VBZ:BE; wordFormStatusInfo: verb, be, simple present, 3sg (bekannt)
Ctrl+Click to follow link

[1] This is a bogus footnote.

Who were the Celts? They arrived in the British Isles around 500 BC. The Romans later called them "barbarians", but they were brave and clever fighters. This is the Battersea Shield, a very famous Celtic shield. You often see circles and shapes like these in Celtic art. But the Celts didn't just leave things behind – some people still speak Celtic languages, like Welsh and Gaelic.

(one small fort for every mile) and bigger forts.

bogus (unbekannt); Adjektiv (bekannt); WKL: ADJ (auch möglich: ADJ:ANA, ADJ:REG, ...); pos: JJ
Ctrl+Click to follow link

[1] This is a bogus footnote.

Figure 8. Tooltips with analysis results for "is" and "bogus" in the footnote. Note that numbers are excluded from analysis

6. Where is the code?

The tokenized-to-tree library is open source and can be found on Github [4]. While the two other applications are proprietary configurations for Ernst Klett Verlag and Deutscher Apotheker Verlag, respectively, the TEI/PDF line-break application will be made available as open source in due time. The repository location will be given in the library documentation, specifically in its README.md document. There is additional inline documentation, as `p:documentation`, in the XProc steps.

7. Conclusion

XProc and XSLT provide both encapsulation and flexibility in complex publishing pipelines, thus enabling re-use.

For performance reasons, it has been suggested that the input be split into smaller processing units.

In order to facilitate re-use, a normalization format for the input chunks has been specified. It provides a side-by-side representation of the normalized input, together with the string-based tokenization/analysis results. This specification is supported by a Relax NG schema.

The three diverse real-life applications demonstrate the suitability and versatility of this library.

Bibliography

[1] Apache OpenNLP: https://opennlp.apache.org/.

[2] Upward projection example: http://www.biglist.com/lists/lists.mulberrytech.com/xsl-list/archives/201407/msg00004.html.

[3] Hub XML: "Conveying Layout Information with CSSa", Proceedings of XML Prague 2013.

[4] https://github.com/transpect/tokenized-to-tree.

Jiří Kosek (ed.)

XML Prague 2018
Conference Proceedings

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and AH Formatter.

1st edition

Prague 2018

ISBN 978-80-906259-4-5 (pdf)
ISBN 978-80-906259-5-2 (ePub)