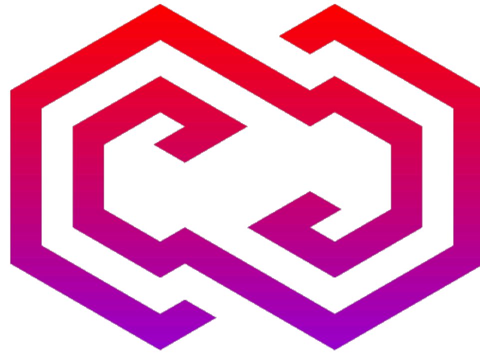
Markup UK 2019 Proceedings

Markup UK 2019 Proceedings



FUSIONDB

SAXONICA .COM
XSLT AND XQUERY PROCESSING

<oXygen/>®
XML Editor

le-tex
publishing services

exeter

MERCATOR

Mulberry
Technologies, Inc.

Organisation Committee

Geert Bormans
Tomos Hillman
Ari Nordström
Andrew Sales
Rebecca Shoob

Programme Committee

Achim Berndzen - <xml-project />
Abel Braaksma - Abrasoft
Peter Flynn - University College Cork
Tony Graham - Antenna House
Michael Kay - Saxonica
Jirka Kosek - University of Economics, Prague
Deborah A. Lapeyre - Mulberry Technologies
Adam Retter - Evolved Binary
B. Tommie Usdin - Mulberry Technologies
Norman Walsh - MarkLogic
Lauren Wood - XML.com

Thank You

Evolved Binary
Saxonica
oXygen XML Editor
letex Publishing Services
Mercator
Exeter
Deborah A. Lapeyre
B. Tommie Usdin
Bethan Tovey
Adam Retter
Jirka Kosek
Norman Walsh
...and our long-suffering partners

Sister Conferences

Balisage
The Markup Conference

summer school

Markup UK 2019 Proceedings

by B. Tommie Usdin, Debbie Lapeyre, Karin Bredenberg, Jaime Kaminski, Peter Flynn, Marco Geue, Gerrit Imsieke, Andy Bunce, Alain Couthures, Andreas Tai, Michael Seiferle, Robin La Fontaine, Nigel A Whitaker, John Lumley, Octavian Nadolu, Tony Graham, Barnabas Davoti, Erik Siegel, Cristian Talau, Liam R E Quin, Syd Bauman, and Sandro Cirulli

Table of Contents

Beyond the brick, for the past in the future, you find the archive! – *Karin Bredenberg. Jaime Kaminski.* 1
Software we have lost – *Peter Flynn.* 11
xprocedit, A Browser-Based Open-Source XProc Editor – *Marco Geue. Gerrit Imsieke.* 39
Generating documents from XQuery annotations – *Andy Bunce.* 47
XQuery for Data Workers – *Alain Couthures.* 57
*sub*check Article MarkupUK London – *Andreas Tai. Michael Seiferle.* 71
An Improved diff3 Format for Changes and Conflicts in Tree Structures – *Robin La Fontaine. Nigel Whitaker.* 87
<Angle-brackets/> on the Branch Line – *John Lumley.* 101
Taking Schematron QuickFix To The Next Level – *Octavian Nadolu.* 125
Accessibility Matters – *Tony Graham.* 135
Scrap the App, Keep the Data – *Barnabas Davoti.* 149
Documenting XML Structures – *Erik Siegel.* 157
XMLPaper: XML-based Conference Paper Workflow – *Cristian Talau.* 171
Dispelling Myths About Markup Formats: When What Why Where – *Liam Quin.* 179
Validating `selector` – *Syd Bauman.* 187
XSpec in the Cloud with Diamonds – *Sandro Cirulli.* 197

Beyond the brick, for the past in the future, you find the archive!

Karin Bredenberg, National Archives of Sweden

Jaime Kaminski, University of Brighton

Abstract

The statement that XML is dead[1] is as wrong as celebrating Christmas on midsummer night's eve! At least in our opinion. Imagine making an archival soup based on international standards using XML, with one municipal archive, two regional archives, five national archives and the European Commission's eArchiving Building block thrown into the mix. This is what we are going to attempt, let us set the stage and take you through the recipe!

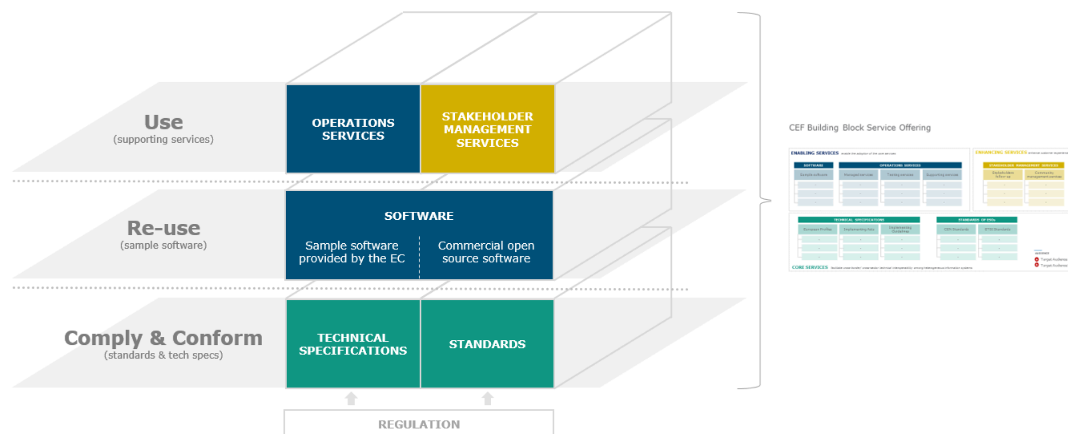
[1]https://developpaper.com/is-xml-dead/ is a starting point for getting more information regarding the statment.

1. Archives

In our world the organisation responsible for saving the records of an organisational unit like documents or other artefacts that gives us our history is the archive. The archives exist in different levels of our society, in companies, in municipalities, in regions and at national levels as national archives. This means that the big difference between a library and an archive is seen in the library being responsible for the printed word. Today when the records are digital records the challenges to keep them has grown from taking care of paper to handling migration of records from obsolete formats, authenticity of the record so its reliable and most of all making sure the record is readable in the future considering format and available hardware.

2. Building Blocks

Let's start with Connecting Europe Facility[2] and its Building Blocks[3], what are they? The European Union realise that internet and digital technologies are transforming our world. A true statement. They also see is that the digital landscape is becoming more diverse, creating challenges for cross-border interoperability and intercommunication. Europe is about working together but, Europeans still face barriers when using (cross-border) online tools and services. The implications are considerable. EU citizens can miss out on goods and services and businesses in the EU miss out on market potential, while also the different governments in EU cannot fully benefit from digital technologies. The EU has therefore described the Digital Single Market[4] (DSM) through which it aims to overcome these challenges by creating the right environment for digital networks and services to flourish. The DSM is not only achieved by setting the right regulatory conditions, but also by providing cross-border digital infrastructures and services. So, to support the DSM, the Connecting Europe Facility (CEF) programme is funding a set of generic and reusable Digital Service Infrastructures (DSI), also known as Building Blocks. The CEF building blocks offer basic capabilities that can be reused in any European project to facilitate the delivery of digital public services across borders and sectors. Currently, there are eight building blocks: Big Data Test Infrastructure, Context Broker, eArchiving, eDelivery, eID, eInvoicing, eSignature and eTranslation. The main part of CEF is a Core Service Platform, provided and maintained by the European Commission. Depending on the building block, the Core Service Platform may include technical specifications, sample software and supporting services (funding for the European Commission). The CEF building blocks offer basic capabilities that can be used in any European project to facilitate the delivery of digital public services across borders. The basis for the CEF Building Blocks are interoperability agreements between European Union member states. The aim of the Building Blocks is thus to ensure interoperability between IT systems so that citizens, businesses and administrations can benefit from seamless digital public services wherever they may be in Europe.



The Building block layers.

For each building block the European Commission provides a Core Service Platform which consists of three layers:

- At the core of each building block is a layer of technical specifications and standards that have to be complied with;
- To facilitate the implementation of the technical specifications and standards, a layer of sample software that complies with them and is meant for reuse (for certain building blocks only);
- To facilitate the adoption of the technical specifications and standards, a layer of services (e.g. conformance testing, help desks, onboarding services, etc.) meant for use (which varies depending on the Building Block).

All this means that the Building Blocks can be combined and used in projects in any domain or sector at European, national or local level.

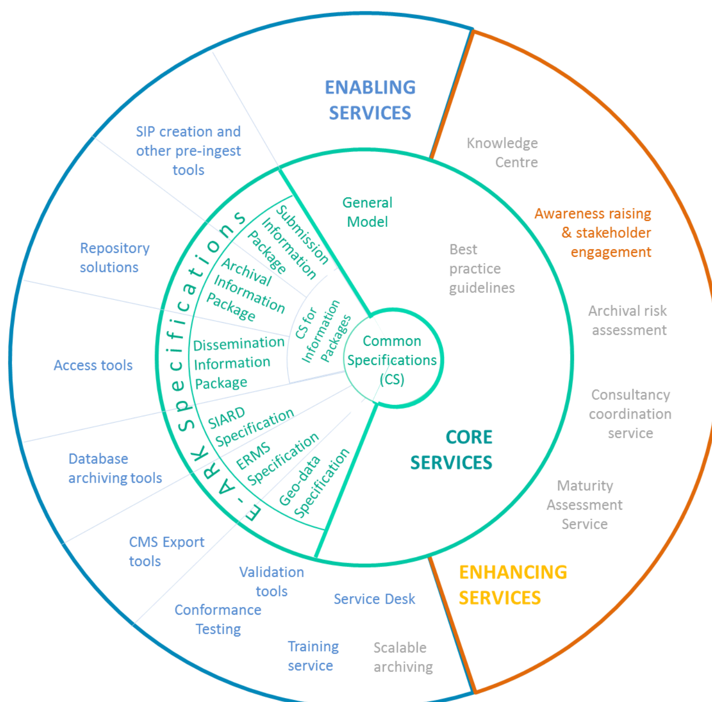
[2]https://ec.europa.eu/digital-single-market/en/connecting-europe-facility

[3]https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/CEF+Digital+Home

[4]https://ec.europa.eu/digital-single-market/en

3. eArchiving Building Block

For the archives and others transferring information the newly set up eArchiving building block [5]is the important component in protecting our history. The aim of eArchiving is to provide the core specifications, software, training and knowledge to help data creators, software developers and digital archives tackle the challenge of short, medium and long-term data management and reuse in a sustainable, authentic, cost-efficient, manageable and interoperable way. The core of eArchiving is formed by Information Package specifications which describe a common format for storing bulk data and metadata in a platform-independent, authentic and long-term understandable way. The specifications are ideal for migrating long-term valuable data between generations of information systems, transferring data to dedicated long-term repositories (i.e. digital archives), or preserving and reusing data over extended (and shorter) periods of time and generations of software systems. Next to the specifications eArchiving offers a set of sample software to demonstrate the format in different scenarios and business environments, and consultancy in regard to long-term digital preservation risks and their mitigation.



The eArchiving building block and its services and specifications.

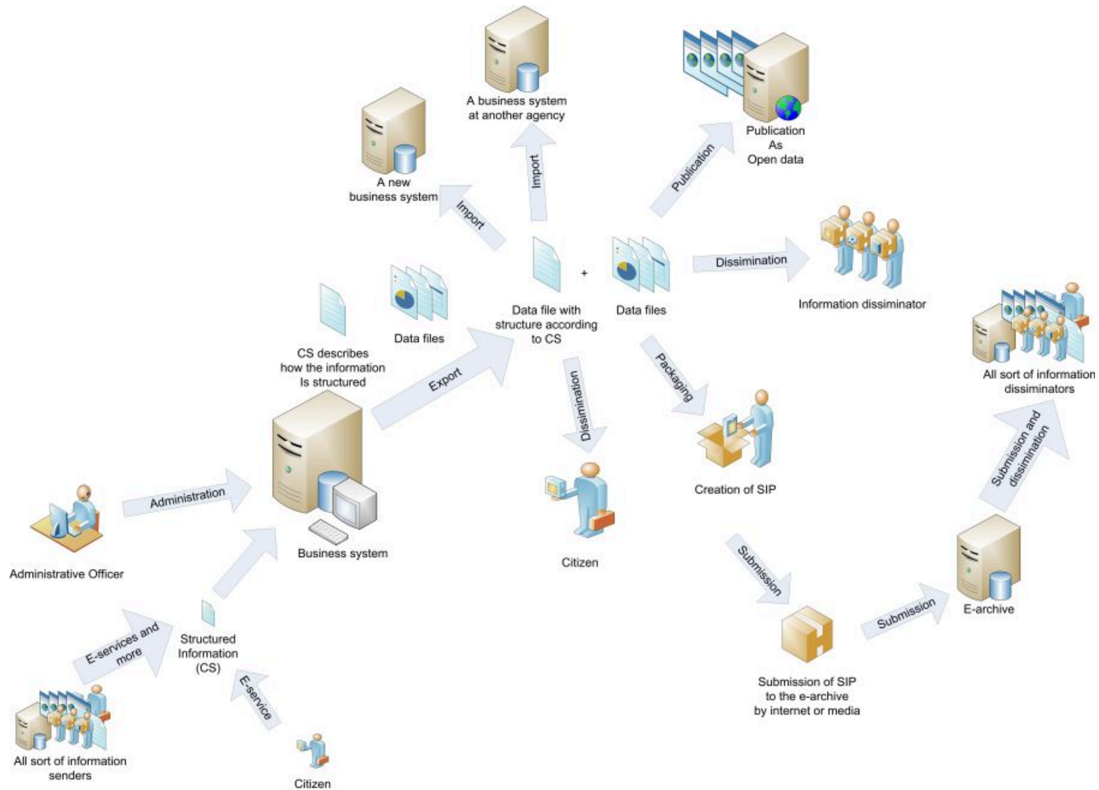
[5]https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/eArchiving

4. Long-term preservation of information

Continuing from the specifications in the building block XML has long been the go-to format for the long-term preservation of information, mainly because it can structure information in a way that isunderstandable to both humans and machines. There have been concerns and comments raised regarding Json taking over the role of XML. However, in the archival setting even if JSon is easier to use as a programmer in the long term XML wins because readable elements and attributes explaining the information or what we call content placed in the document. However, at the same time XML needs to be used wisely with the correct element and attribute names to facilitate the understanding of the human reading of the XML-document both now as well as in the future. There are also other formats used like tiff and pdf but for reuse of the information XML is the format to go to in most cases.

5. Use cases

Our use case involves transferring information to different recipients. Information described with numerous ISO as well as *de facto* standards, all built around XML, which can be used for the transfer, storage and dissemination of information. Some of the most important use cases are shown in the following image.



The eArchiving use cases.

A short description of what we see in the image is that information is created in a system, the information is being exported as a number of different documents which can be sent to an electronic archive, being used by a researcher, published online, imported into a new system and so on.

6. Skill set

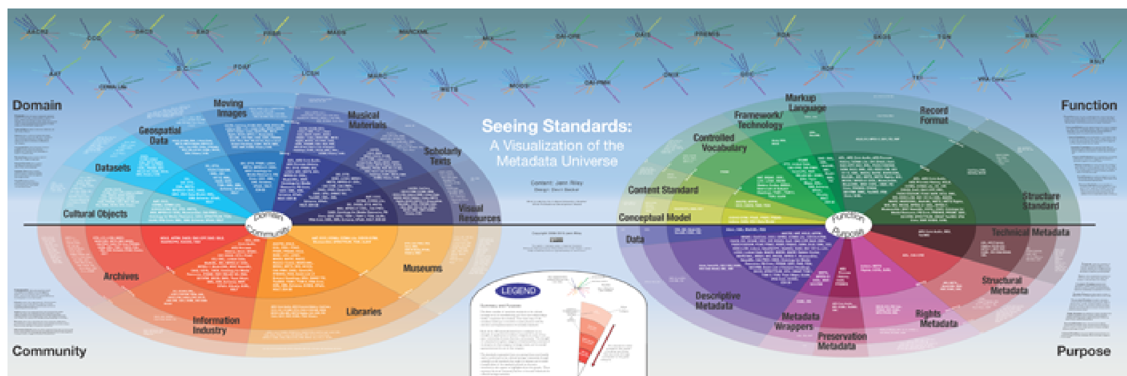
And with that setting of uses cases and need for reuse in a lot of ways we can only make the conclusion that a modern archivist, or perhaps the more accurate title is information or records manager, needs to understand XML and more than one standard for handling the information for which they are the caretakers. Besides the need for knowing a lot of different technical languages there is also a need to speak the same language as the programmers who will aid you with creating export and import tools. A challenge still not fully addressed in either community. The language challenge is something all the different communities and professions working together need to start working with. A common solution is that every work place and profession has their own language making cross-border professional exchange almost impossible without a lot of extra meetings, things that can be avoided if a common vocabulary was agreed upon and communicated to all professions. Look, for example, at the term 'archive' which means "to transfer records from the individual or office of creation to a repository authorised to appraise, preserve, and provide access to those records"[6] in the archival setting and the meaning "to store data offline."[7] in the programmer and more technical setting. A unification is needed and at the same time not easy to achieve due to the different professions lack of common foras for these kinds of discussions (let's not forget that not speaking to each other is more common than we would like to imagine).

[6]https://www2.archivists.org/glossary/terms/a/archive

[7]https://www2.archivists.org/glossary/terms/a/archive

7. Standards, de facto standards and specifications

Jenn Riley's Visualization of the Metadata Universe[8]has been around for a while, but still gives the best overview of standardsused in the cultural sector, archives, libraries and museums. In the image there are numerous standards are displayed in the context of their function and where they are used.



The visualization of the Metadata Universe by Jenn Riley. (The recommendation is to look on-line)

Almost all standards on the metadata map created by Jenn Riley have an XML format available described with an DTD or an XML-schema. For the XML-schemas both the ISO standard RelaxNG as well as W3C XML-schema formats are used. The choice depends solely on the skills of the creator of the schema and at the same time it's also common to ensure that all different type of schemas are available so transformations from RelaxNG to XML-schema and vice versa is often used. DTD are still around due to the fact that old software is still in use and they are based upon using a DTD.

The most common way to use the standards is to write a specification which describes a profile for our use case of the standard which then in its turn are implemented in the setting you are operating. The best way of describing the use of profiles is the standard METS (more about that later) which requires the user to write a profile describing how it is used.

[8]http://jennriley.com/metadatamap/

8. eArchiving Building block Specifications

The story of eArchiving Building Block specifications originates at work starting at the National Archives of Sweden and being enhanced in the E-ARK project[9]. The project delivered 7 draft specifications which all built upon creating profiles of standards and de-facto standards, these can be split into 4 specifications describing profiles of the standard METS and one for electronic records management, one for geodata and finally one for using the SIARD [10]standards. The project ended but no one wanted the work to be a usual project result, forgotten and not used so to make sure these specifications to not stop evolving the project created the "Digital Information LifeCycle Interoperability Standards Board" (DILCIS Board[11]). The board took over the specifications and have during the project E-ARK4ALL[12] (which are responsible for setting up and maintaining the eArchiving Building Block) brought them to a stabilised state and started the development of more specifications. A work carried out together with the experts of specific content so it's the experts writing the specifications. The board is set up with currently eight members and are charged with the task of handling the specifications. The Board is at the same time the core producers of specifications to the eArchiving Building Block.

[9]https://eark-project.com/

[10]https://www.bar.admin.ch/bar/en/home/archiving/tools/siard-suite.html

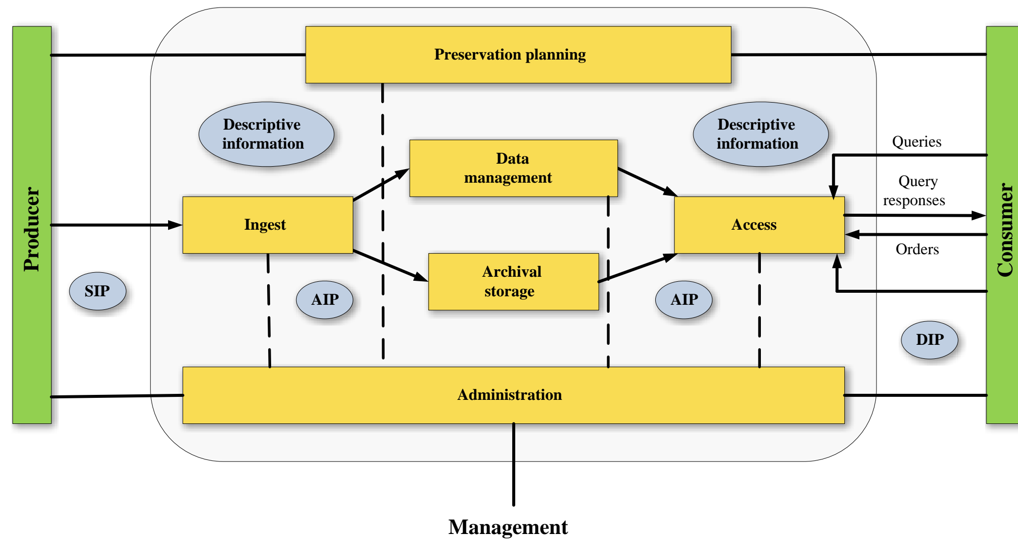
[11]https://dilcis.eu/

[12]https://e-ark4all.eu/

9. The eArchiving reference model setting

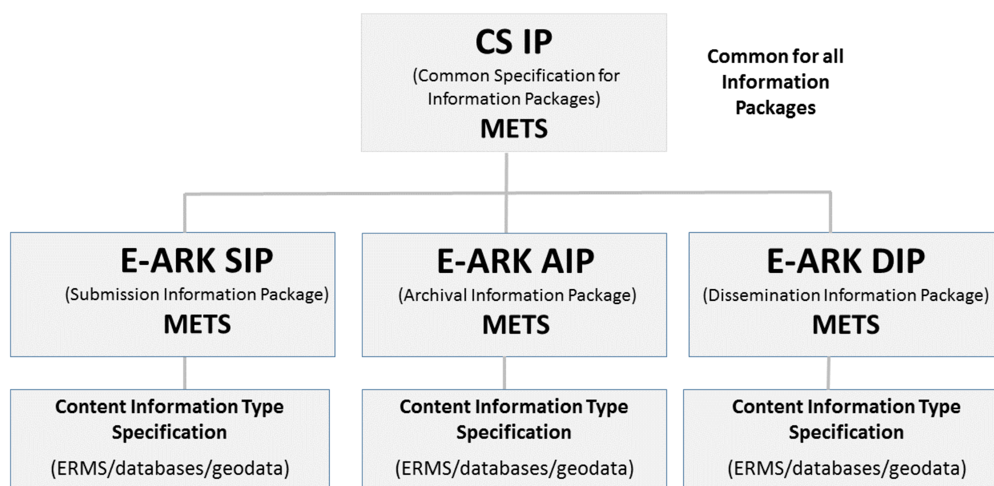
When we consider the European Commission's new eArchiving Building Block and that its evident that at its core are specifications based upon formats expressed in XML. The entireBuildingBlockrevolves around transfers of information where the final transfer is to the archives, but nothing prevents these transfers occurring earlier in the information lifecycle. The e-archive is built upon the Reference Model for an Open Archival Information System (OAIS Reference Model)[13]and its use of information packages, the Submission Information Package (SIP), the Archival Information

Package (AIP) and the Dissemination Information Package (DIP). There are more parts described in the reference model but the core part used in the specifications are the information packages.



The OAIS reference model.

The different packages in the OAIS reference model are within the eArchiving Building Block described or get their inventory or manifest stated with the Metadata Encoding and Transmission Standard (METS)[14] which uses XML as the format for creating the readable text. (For the AIP it might be a format internal to the archival system you are using but that is another paper.) This way both machines and humans can understand the package (we do have some extra principles which aid with what constitutes a package). METS itself is a rather open standard with only one mandatory element being a structural description of the package. The standards also demand the creation of a profile for the exact use case describing the use of the standard and its elements and attributes. The profile for the eArchiving Building Block requires besides the METS XML-schema an extension XML-schema and validation rules in Schematron. The difficulties occur when the common user doesnot know how to validate the XML in combination with Schematron due to poor or no knowledge of either XML or Schematron and not having access to a person with the knowledge. We must not forget that Schematron adds its owncomplexities when it is combined with a non-relaxNG schema. A complexity that can be overwhelming when you don't know XML at all or just a basic understanding. This gives that numerousguidance documents need to be created ranging from how to write XML to how to use a specification. And all this since you cannot count on the person implementing the specification and its validation to have the appropriate background knowledge.



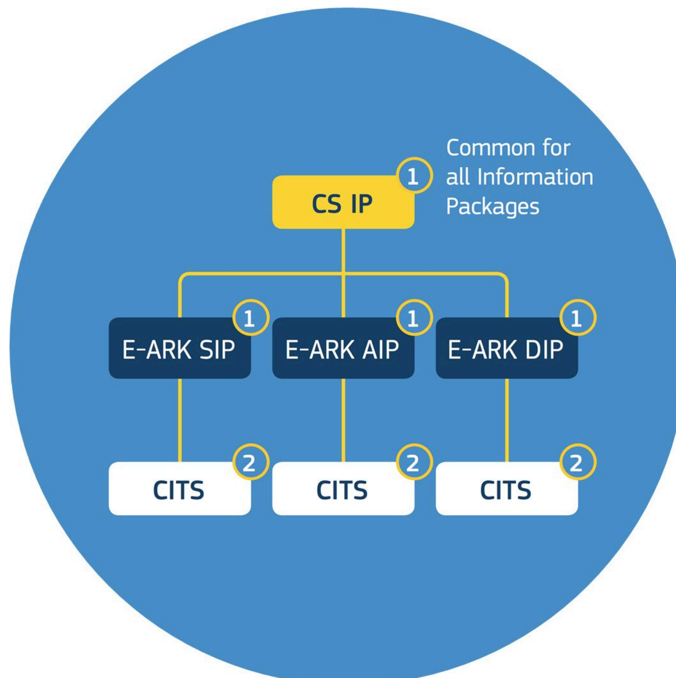
The eArchiving building block and the different specifications building up a Information Package. In the image the three different types of packages is seen.

[13]https://www.iso.org/standard/57284.html

[14]http://www.loc.gov/standards/mets/

10. Content specifications

Well, we now have a package described in XML, we need some content and maybe images, PDFs and information structured in XML that extends the information recorded in the package. This means we have one or more XML-documents to describe the information and its structure, and another XML-document to describe the whole package with all the content, but not the structure of the information itself. We can even handle a relational database in this way, extracting it in XML format and then packaging it in a SIP for transfer to the archive. Specifications of content are at the heart of the eArchiving Building Block, and the number of specifications is growing steadily. There are lots of different kinds of information that need to be described, luckily there are many specifications for describing information in XML, so there is no need to reinvent the wheel.



The eArchiving building block specifications.

In the image the content is described with the acronym CITS which means Content Information Type Specifications. Currently there are three available as described previously.

- A specification for electronic records management systems (ERMS), the specification uses a XML-schema as the format and is based upon several available records management standards which in their turn don't have a common XML-format available which means the ERMS specification is the connection between the different standards and the export of information from a ERMS.[15]
- A specification for geospatial data which uses the ISO standard for preservation of geospatial data in combination with the regulation within the union regarding geospatial data "the Inspire directive". In the specification a description of geospatial data and what it is found together with how the description of the data is carried out since the geospatial system themselves export information in readable formats but lack descriptions making the information understandable in the future.[16]
- A specification for how to place a relational database exported with the format SIARD in an information package. The format has been developed by the Swiss Federal Archives and is now a part of the DILCIS Boards responsibilities. The SIARD format is based upon export of the database as XML and several tools exist which aids with the task.[17]

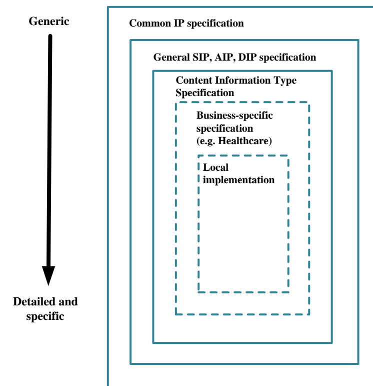
[15]https://github.com/DILCISBoard/E-ARK-ERMS

[16]https://github.com/DILCISBoard/E-ARK-Geodata

[17]https://github.com/DILCISBoard/SIARD

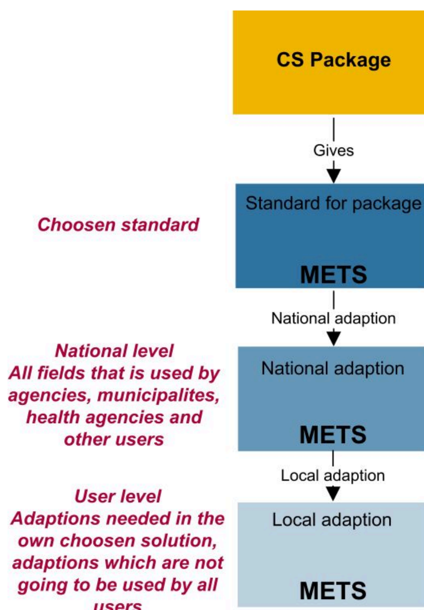
11. Basic soup recipe with a twist

We started with you imagine the making of an archival soup based on international standards using XML, with one municipal archive, two regional archives, five national archives and the European Commission's eArchiving Building block thrown into the mix. So how do we get this soup to be the "perfect" soup? We use core specifications based upon XML that can be enhanced the further down or closer to the user you get thus making the work and transfer to an archive a piece of cake.



The specification enhancement layers.

So our soup gets transformed into the set of use cases which is suitable for different kinds of users as well as different kinds of information.



The specification user layers of adoption.

The image shows us that the closer to the information and the implementation you get the more demands is added to make the information understandable in the future. We still are obliged to use the core set which means interoperability is achieved and at the same time the users own needs is an addition to enhance the core set.

12. Conclusion: Moving on from soup

The archival soup isn't in fact a soup it's a core set of common specifications using a machine and human readable language which makes it possible for one municipal archive, two regional archives, five national archives and the European Commission's eArchiving Building block to transfer information, store the information for "infinitive" time and deliver it to all different kind of users wanting to use the information, use the information for researchers, building applications, doing statistical reposts, prove who they are, all the things you can do with information. This is supported by the advantages of XML and its way of being readable both by machine and humans. That the XML-document can be opened in just a text editor means that it is easy to read our past both now and in the future. This means that the support for XML needs to exist now and further down the road even if we in the archival community as XML users do not take part in all working groups maintaining XML instead being more concerned about the standards, *de facto*standards or specifications based on XML so we can ensure the past being created now will be present in the future.

Bibliography

[MDU] Jenn Riley: Seeing standards: A visualization of the Metadata Universe. Design: Devin Becker2009-2010, Work funded by the Indiana University Libraries White Professional Development Award. http://jennriley.com/metadatamap/

Software we have lost

Peter Flynn

Abstract

Since the first days of SGML, there has been a variety of software to parse, validate, analyse, format, store, search, and extract the information. Some of this was what we now call Open Source, particularly the smaller utilities, but the majority of applications were conventional commercial offerings.

In the course of time, many of these have become unavailable, for assorted reasons, with the result is that some very useful systems have been lost, and replacements are not always as effective.

This research attempts to catalogue and analyse a collection of XML and SGML software that is either off the market, or only available within a different product, and thus not accessible to users. The objective is to see if there are still ways to shorten the distance between the bricks that are not otherwise provided for.

I am grateful to the numerous people in University College Cork and elsewhere who stepped up with offers of Windows and other installer CDs when my carefully-preserved originals went missing, including (alphabetically) John Barrett, Roy Cummins, Stephen Dineen, AV Drepe, Martin Fleming, Nick Hogan, Sinead Horgan, Steve King, Margaret Lantry, Piaras MacEinri, Neil Nash, John O'Connell, Michael O'Halloran, Billy O'Rourke, Bereniece Riedewald, Joel Walmsley, and Frank van Pelt. Thank you also to the SGML-era veterans who prompted me with the names, details, or disks of long-forgotten products, especially Debbie Lapeyre, Lauren Wood, and Michael Sperberg-McQueen

1. Background

Before we had XML, we had SGML, which is the ISO standard techreport on which XML is based. By default, SGML is pointy-bracket markup like XML…but in SGML almost everything can be redefined, including the markup characters themselves, and there are numerous options for additional features and for abbreviations and markup shortcuts to minimise typing. Any changes to syntax or to the configuration value limits have to be made in a Declaration file, and a DTD is compulsory on every document.

Notice the words to minimise typing. In the beginning, there was no software with an editing interface that could use the DTD to provide a contextual menu of available element types; and the idea that the markup would be hidden from the user was counterintuitive — if you couldn't see the tags, how could you know what was marked?

In these early stages, therefore, markup was applied by typing it in a plaintext editor, so *the* essential piece of software to begin with was the parser, not the editor, so that you could check that you hadn't got something wrong. The term *parsing* was often used to mean parsing-and-validating; [1] as there was no concept of well-formed tag validity in the sense introduced with XML. There were many early parsers; among the most significant were:

- ARC SGML (Almaden Research Center) originally by Charles Goldfarb; later developed by James Clark into sgmls (see below)
- ASP SGML (Amsterdam SGML Parser), still available article
- Exoterica by Sam Wilmott (later included in Omnimark)
- the parser in Framemaker+SGML by Lynn Price
- a parser for Boeing (internal only) by Greg O'Connell and Debbie Lapeyre
- Mark-It! by Jean-Pierre Gaspard
- sgmls by James Clark, the only one still in widespread use; redeveloped as nsgmls for SP, and now as onsgmls to handle XML for OpenSP

Other software developed rapidly, spurred partly by the adoption of SGML for some military documentation in the US and elsewhere, and partly by its growing use in publishing, research, and academia. Editing software included:

- Arbortext ADEPT (through several name changes (eg Epic), now PTC Arbortext Editor)
- SoftQuad Author/Editor and the editors based on it, HoTMetaL (for HTML) and later, XMetaL (for XML)
- STiLO Document Generator, with Arbortext one of the few to handle mathematics in a general-purpose SGML editor
- Emacs with psgml-mode
- epcedit, a free SGML and XML editor from tkSGML
- the Euromath Editor, an EU project built on the GriF editor[2]
- Siemens Nixdorf InContext
- Citec MultiDoc Translating Editor
- Microstar Near&Far Author for Word and Near&Far Designer, a graphical DTD editor
- GriF SGML Editor
- Richard Light's SGML Tagger (OUP), a memory-resident monitor for MS-DOS editors.
- Corel WordPerfect had a built-in SGML editor
- Sema Write-It! (using Mark-It! as the parser)

Documents also need processing in some way: adding to a database, putting on the Web, mining it for data, or converting it for a formatting system for publishing. Conversion or processing (transformation) systems included:

- AIS Software Balise
- DFN DAPHNE (VMS only; converted to TeX)
- EBT (later Inso) DynaText trainable converter from Word to SGML
- James Clark's Jade (using DSSSL) can convert to TeX and other formats
- Exoterica Omnimark (XTRAN)
- Microsoft SGML Author for Word, despite its name, this was *not* an editor, but a converter into and out of Word

[1] In fact, in the authors' description of the Amsterdam SGML Parser article, the only instances of the term *validation* are in the formal references to validation *services* in the SGML standard itself.
[2] The editor is reputedly being resuscitated and rebuilt using INRIA's Thot structured editor.

There were several standalone viewers, especially for vertical-market applications, but few general-purpose browsers. As with editors, some used SGML-syntax stylesheets to format the display; others used proprietary stylesheet syntax. Formatting systems for printed output typically produced Postscript (pre-PDF days). Some handled SGML input direct, others via an established conversion route; output was formatted using TeX or a proprietary typesetting engine.

- Advent 3B2 typesetter
- EBT DynaWeb NT server for documents converted with DynaText
- Adobe Framemaker+SGML typesetter (FTC's original had no SGML support)
- LaTeX, typesetter, usually via transformation through Omnimark, Balise, Jade, or similar
- Citec MultiDoc Pro Publisher standalone browser
- Panorama Viewer, an SGML plugin for the Mosaic and Netscape browsers; also the standalone Panorama Publisher
- Arbortext Publisher typesetter

There was far more software available which is outside the scope of this report — some of it is now either uncompilable or uninstallable, or was in any case incomplete or experimental at the time. A significant amount was normal commercial software which has suffered the conventional fate of being superseded, falling out of use, or being abandoned when the company failed or was taken over. There are extensive lists of both free and commercial applications in Robin Cover's SGML/XML Web Pages [http://xml.coverpages.org/index.html], and some of the SGML Conference CDs have a considerable amount of freely-distributable and commercial-sample software in subdirectories..

Other categories not covered here include design tools, search engines, and databases. The only three of these of which this author has direct experience (noted below) were Microstar's Near&Far Designer, Tim Bray's PAT search engine in Section 2.5.2 [31], and the SGML DARC document management database in Section 2.5.3 [31].

2. Software

The sections below refer to those programs and systems which were *either* installed and [re-]tested for this paper *or* were tested and documented in the author's book book. The following symbols are used:

- a checkmark beside an item denotes that it installs and executes correctly

- an X denotes that the software exists but cannot be compiled or installed correctly, so testing it was not possible

- a circle denotes that the software could be installed but either would not execute, or executed but with unresolvable errors

- an empty box denotes the software is no longer available

The platform used for testing Windows and MS-DOS software was Windows XP SP2 running on a Dell Optiplex 745.[3] The objective was to emulate as reasonably as possible the office environment *circa* 1998–2002. A modern Linux distribution (Mint 19) was used for the few UNIX or GNU/Linux utilities. The procedure for testing was:

1. Install the software from original media where possible, or from zip archives from network repositories

2. Run the relevant program[s] from the Start menu or from an installed icon (a few command-line procedures were run from the Windows Command terminal)

3. Open or otherwise invoke the sample SGML document (see Appendix A [34]), performing any necessary prerequisites such as making the DTD or SGML Declaration available to the program

4. Exercise the features or functions of the software to check they operate correctly (eg Insert Element, Edit Attribute, Validate, etc)

5. Record details of success or failure

2.1. Parsers and validators

Three of the products listed in ???TITLE??? [12] were tested.

2.1.1. ARC SGML

This software was installed from the copy distributed on the SGML'97 CD-ROM. However, the `INSTALL` and `readme` files referenced binaries that were not included, and attempting to parse any of the test suite resulted in

[3]An attempt was made to use Windows 95 but this satisfied the requirements for only the oldest programs.

references to components that were not recognised. However, the `vm2` program did appear to execute, but failed to parse the sample file, returning error messages related to capacities, and it was not clear how the values from the SGML Declaration could be implemented within the time available.

2.1.2. ASP SGML

Several attempts were made to compile this parser from the source code available in `Sgml.tar.z` but the changes in the C libraries over the years mean that significantly more work is needed to recode those parts which currently generate errors. While this would be an interesting excursion for a student coding project, there is probably little to be gained by resuscitating the software for production use when onsgmls fulfils the same function.

2.1.3. sgmls

Derived from ARC SGML, the parser was rewritten as part of the Jade DSSSL processor (now OpenJade). This is a standalone (commandline) parser with options (among others) for:

- suppressing the default ESIS output (a markup-and-data stream) when just a validity check is required
- limiting the number of error messages
- switching to XML mode (onsgmls only)
- specifying the SGML Declaration to use

sgmls is also used by the sgmlnorm normalizer to expand SGML shortcuts; and by the Emacs/psgml editor for validation. The 32–bit Windows binaries used in this test were installed from the SGML'97 CD-ROM.

```
C:\SGMLS\> sgmls -s sgml.dec recipe.sgml
```

In the case of this, and most command-line parsers, a null return means no errors were found in the DTD or document.

The companion sgmlnorm utility was also exercised: the normalization fills in all the missing parts of the SGML document by applying the rules from the SGML Declaration on what end-tags may be omitted and if attributes may be minimized (and much else). In this case the following output was obtained:

```
C:\Documents\> sgmlnorm sgml.dec recipe.sgml
<RECIPE>
<TITLE>Chocolate fudge</TITLE>
<COMMENT>My mother's recipe</COMMENT>
<INGREDIENTS>
<INGREDIENT QUANT="1" UNITS="LB">sugar
    </INGREDIENT>
<INGREDIENT QUANT="4" UNITS="OZ">chocolate
    </INGREDIENT>
<INGREDIENT QUANT="?" UNITS="PT">cream
    </INGREDIENT>
<INGREDIENT QUANT="1" UNITS="OZ">butter
  </INGREDIENT>
</INGREDIENTS>
<METHOD>
<LIST>
<ITEM>Mix the ingredients in a pan
      </ITEM>
<ITEM>Heat to 234?F, stirring constantly
      </ITEM>
<ITEM>Pour into greased flat tin
      </ITEM>
<ITEM>Allow to cool before cutting
    </ITEM>
</LIST>
</METHOD>
<SOURCE>Adapted from the Good Housekeepings cookbook</SOURCE>
</RECIPE>
```

Note that by default, SGML is case-*in*sensitive, and that the pretty-print indentation of the original document means that normalizing the omitted end-tags has introduced a white-space node into the character data content of the `ingredient` and `item` elements.

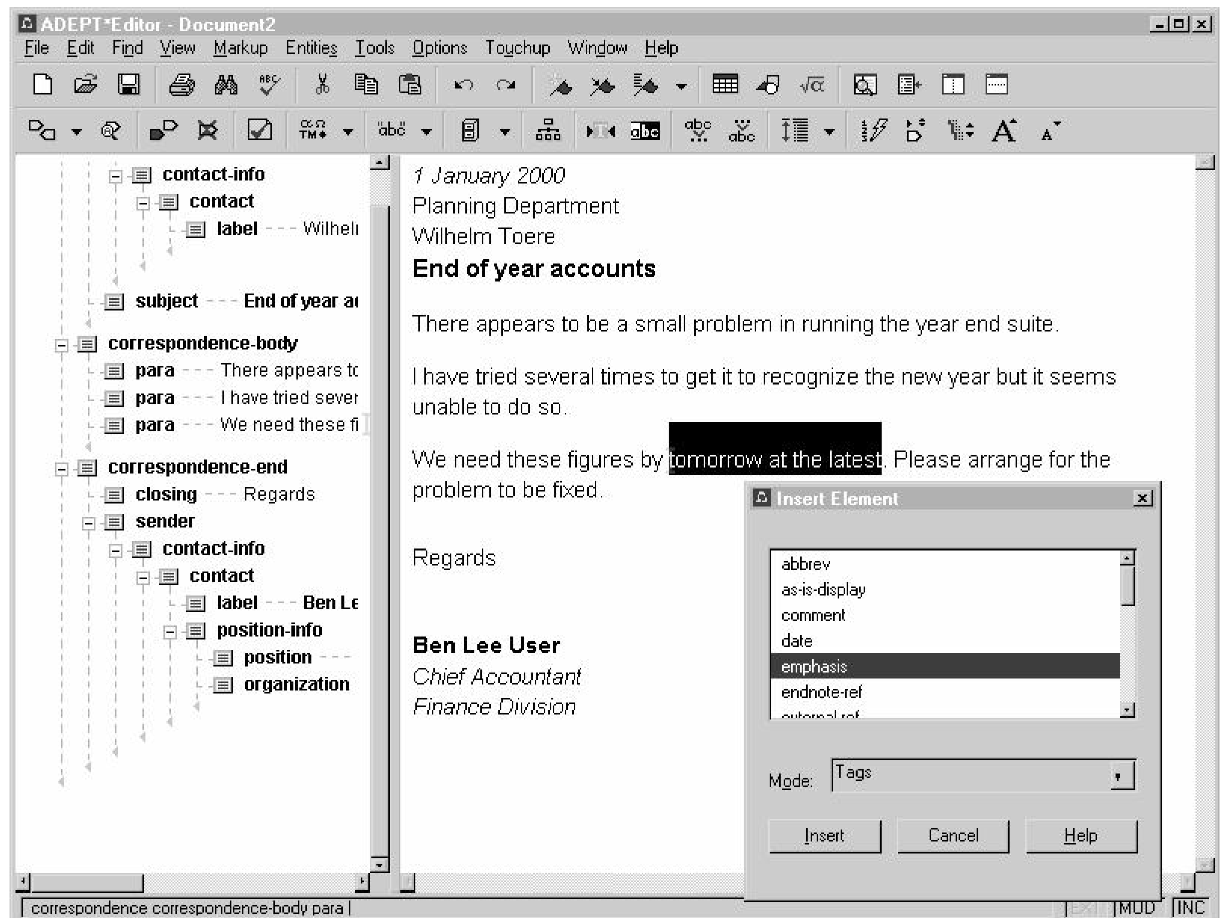
2.2. Editors

Of the editors in ???TITLE??? [12], Document Generator, SGML Tagger, Euromath Editor, and Write-It! were not reviewed either due to lack of software or incompatibilities.

2.2.1. ADEPT

The Arbortext editor is perhaps the best-known industrial SGML and XML editor. It was (is) a very large suite, including the Document Architect program for compiling DTDs and FOSI stylesheets, and Publisher which output typeset-quality formatting (printing from the editor itself was more office-quality). Setting up the suite takes a long time, especially if many custom DTDs and stylesheets need to be installed: this is not a simple editor but a fairly complete document production system — the Author/Editor and Panorama Publisher setup had similar capabilities, but the editor could also be installed and running in a few minutes, which is not the case with ADEPT. It was also *expensive*: in 1992, the present author was quoted $5,000 per seat and no discounts.

Figure 1. ADEPT showing the Insert Element dialog (left); and editing mathematics (right)



A strong feature of ADEPT was mathematics (in an earlier incarnation, Arbortext also sold a commercial version of TeX, and the TeX engine remained inside their products for many years). The graphical interface to mathematical editing (see Figure 1 [15]) was a *de facto* industry standard which was only challenged by LyX and more recently Word's Equation Editor.

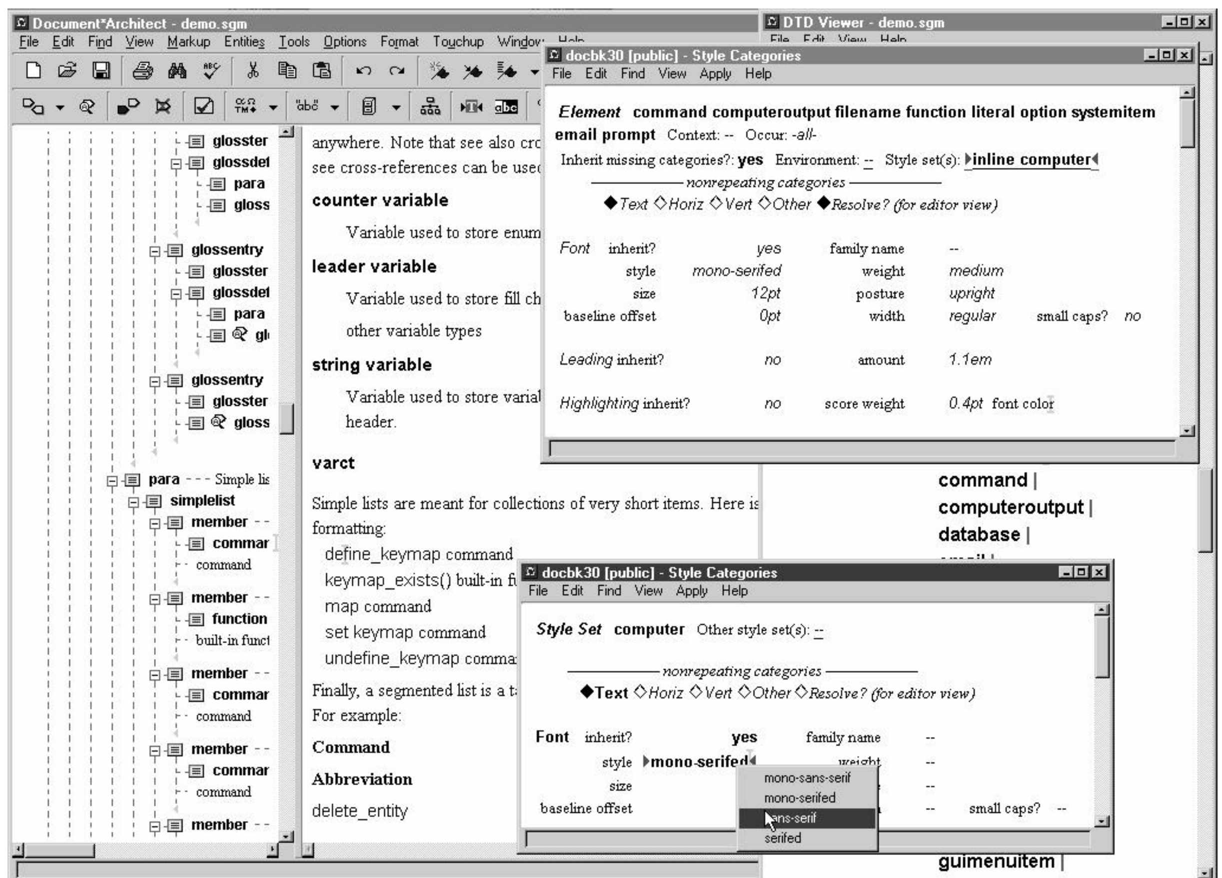
As with Author/Editor and RulesBuilder, the principle was to separate the business of editing — the job of professional technical writers — from the business of maintaining the DTDs, many of which were from industrial vertical markets

and subject to strict data controls over who could change what and when and how. One interesting approach, not seen in any other product of the era, was that after a DTD was successfully compiled for the first time, the system would ask for the element type names used for:

- the document title, title block, and title page
- normal paragraphs
- graphics, and attribute names for assorted graphical manipulation features
- divisions (chapter, section, subsection, etc)
- lists (numbered/bulleted/definition)
- figure blocks and page breaks

This was then used to construct an initial FOSI stylesheet so that when a new instance of the document type was created or opened, it would at least be styled with the basic structure rather than presented as an amorphous slab of markup and text. The stylesheet could then be edited to refine the formatting (see Figure 2 [16]): the styling interface was comprehensive but required substantial training to use.

Figure 2. Document Architect creating styles for a document type



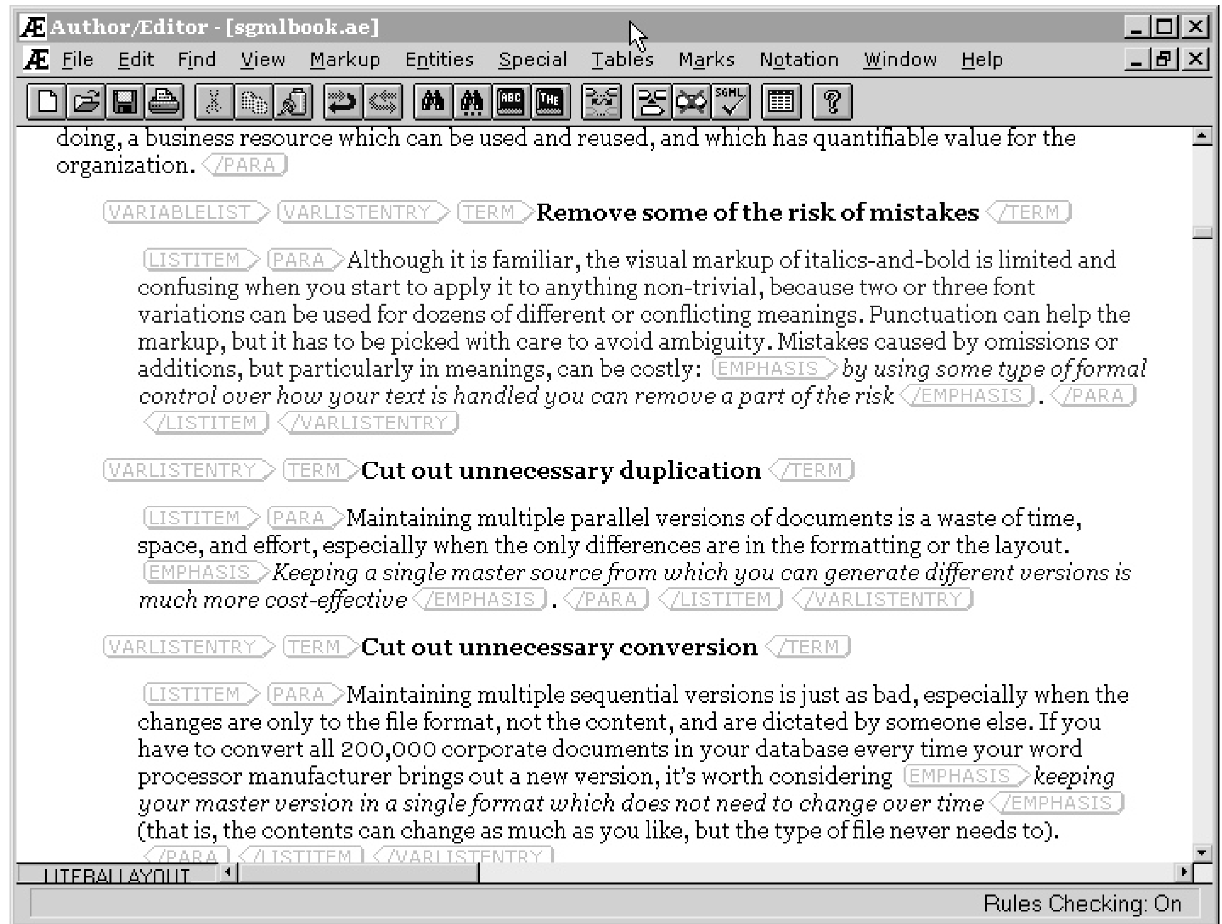
2.2.2. Author/Editor

RulesBuilder #

This was one of the most widely-used editors: easy to use and fuss-free in operation. Its interface later became familiar to a much wider audience as it formed the basis for the HTML editor HoTMetaL and the XML editor XMetaL. Installation was from CD.

Author/Editor would immediately open an arbitrary SGML file, but would only enter full synchronous typographic mode if the DTD was known and precompiled. For documents using previously unencountered document types, it acted as a plaintext editor until provided with a compiled DTD.

Figure 3. Author/Editor editing a DocBook document (left); RulesBuilder compiling a DTD (right)



For full use, the separate RulesBuilder program was needed. The assumption was that the DTD was a corporate asset that would not be available to an end-user, only to an administrator; and even outside that type of managed framework, it would be undesirable for arbitrary users to be able to modify what should be a stable DTD. The administrator would compile the DTD to a proprietary `.rb` file which *could* be given to an end-user.

2.2.3. Emacs + psgml

GNU Emacs is a plain-text editor and work environment by Richard Stallman, with a macro and programming facility using a dialect of Lisp, used extensively for the creation of add-on packages. Emacs 20.7 for Windows 95 and NT was installed from the GNU archive [ftp://ftp.gnu.org/old-gnu/emacs/windows/20.7/].

The psgml-mode package by Lennart Staflin is an Emacs major mode for SGML and XML which parses the DTD (a requirement in SGML). At the time it provided the only free and unencumbered fully-featured SGML editor.[4] Installation of various versions of psgml were tested to find one which was compatible with the 20.7 version of Emacs, as at that time there was no package library synchronisation. Version 2.12 worked with the removal of the compiled form of the `psgml-other.elc` file, which was the only code mismatch. Installation would normally be via the Emacs package system,[5] or by using the `Makefile` in a downloaded psgml distribution, or (in the test case under Windows) by copying the files to a suitable directory manually.

[4] Arguably, it still does, and for XML as well.
[5] The package has recently and inexplicably been withdrawn from the current repositories.

Figure 4. Emacs with the sample document before and after normalization

emacs@CALYX

File Edit Options Buffers Tools SGML Help

```
<!doctype recipe system "recipe.dtd">
<recipe>
  <title>Chocolate fudge</title>
  <comment>My mother's recipe</comment>
  <ingredients>
    <ingredient quant='1' lb>sugar
    <ingredient quant='4' oz>chocolate
    <ingredient quant='&frac12;' pt>cream
    <ingredient quant='1' oz>butter
  </ingredients>
  <method>
    <list>
      <item>Mix the ingredients in a pan
      <item>Heat to 234&deg;F, stirring constantly
      <item>Pour into greased flat tin
      <item>Allow to cool before cutting
    </list>
  </method>
  <source>Adapted from the Good Housekeepings cookbook</source>
</recipe>
```

U:--- recipe.sgm All L1 (SGML Fill)

```
<!doctype recipe system "recipe.dtd"[
]>
<RECIPE><TITLE>Chocolate fudge</TITLE>
<COMMENT>My mother's recipe</COMMENT>
<INGREDIENTS>
<INGREDIENT QUANT="1" UNITS="LB">sugar</INGREDIENT>
<INGREDIENT QUANT="4" UNITS="OZ">chocolate</INGREDIENT>
<INGREDIENT QUANT="½" UNITS="PT">cream</INGREDIENT>
<INGREDIENT QUANT="1" UNITS="OZ">butter</INGREDIENT></INGREDIENTS>
<METHOD>
<LIST><ITEM>Mix the ingredients in a pan</ITEM>
<ITEM>Heat to 234°F, stirring constantly</ITEM>
<ITEM>Pour into greased flat tin</ITEM>
<ITEM>Allow to cool before cutting</ITEM></LIST></METHOD>
<SOURCE>Adapted from the Good Housekeepings cookbook</SOURCE></RECIPE>
```

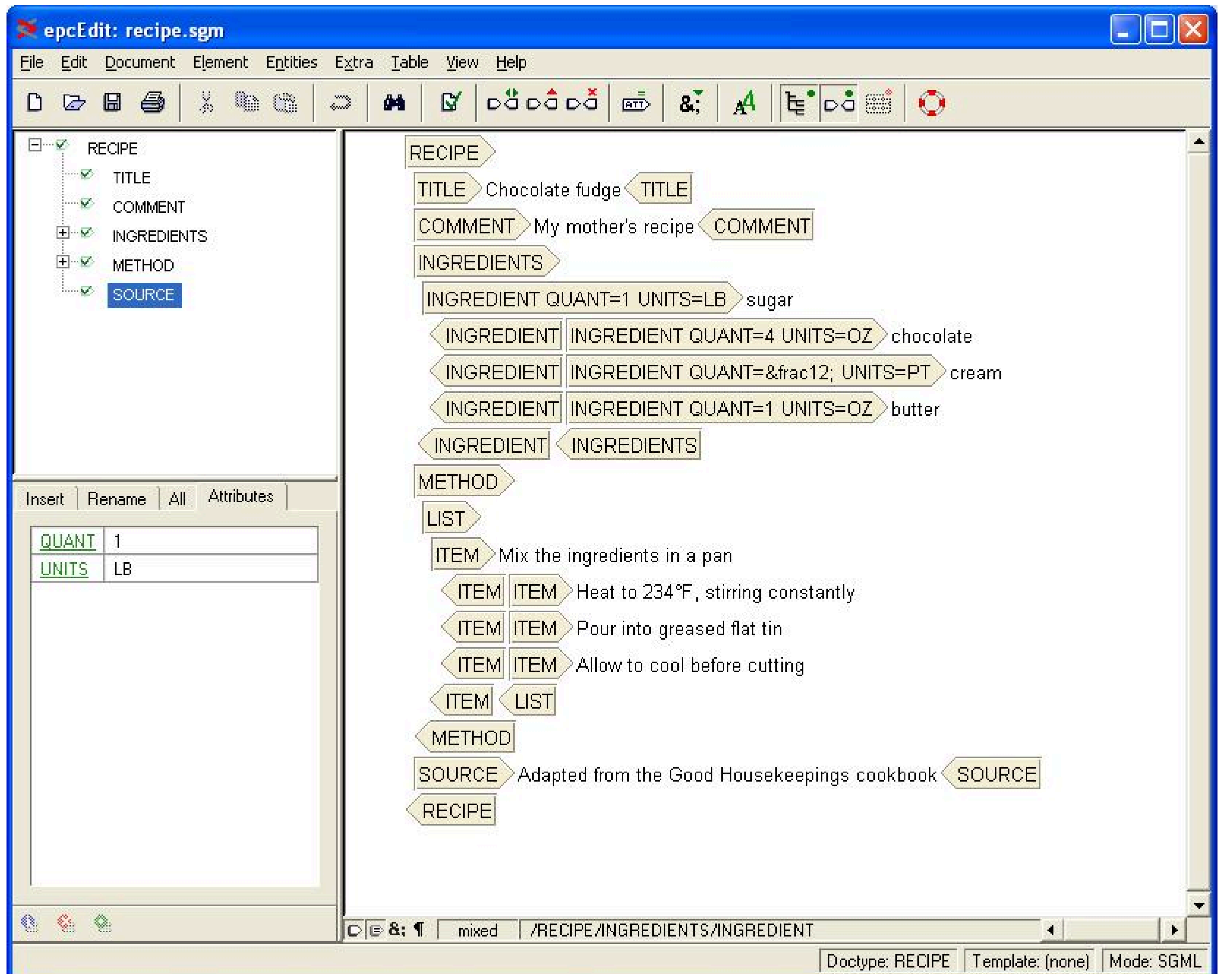
1:**- recipe-normalised.sgm All L2 (SGML Fill)

A setting in the user's `.emacs` configuration file can make Emacs invoke psgml-mode automatically for files ending in `.sgml` or other extensions. The DTD is tokenised and the result used to guide the user's selection of element types via a menu or with TAB completion, and the use of attributes via a subsidiary window. Manipulation of markup in context as with any other syntax-directed editor is done from the menus or with keystroke abbreviations.

2.2.4. epcedit

This is an SGML (and later, XML) editor from tkSGML by Heinz Detlev Koch and Roman Halstenberg. While it is no longer under development, it can still be downloaded but it is a 32–bit system only and requires a licence key. The program uses the Tcl/tk language (v8.3) which is included in the distribution. The sample document was opened without problems (see Figure 5 [19])

Figure 5. epcedit with the sample document and with the sample stylesheet

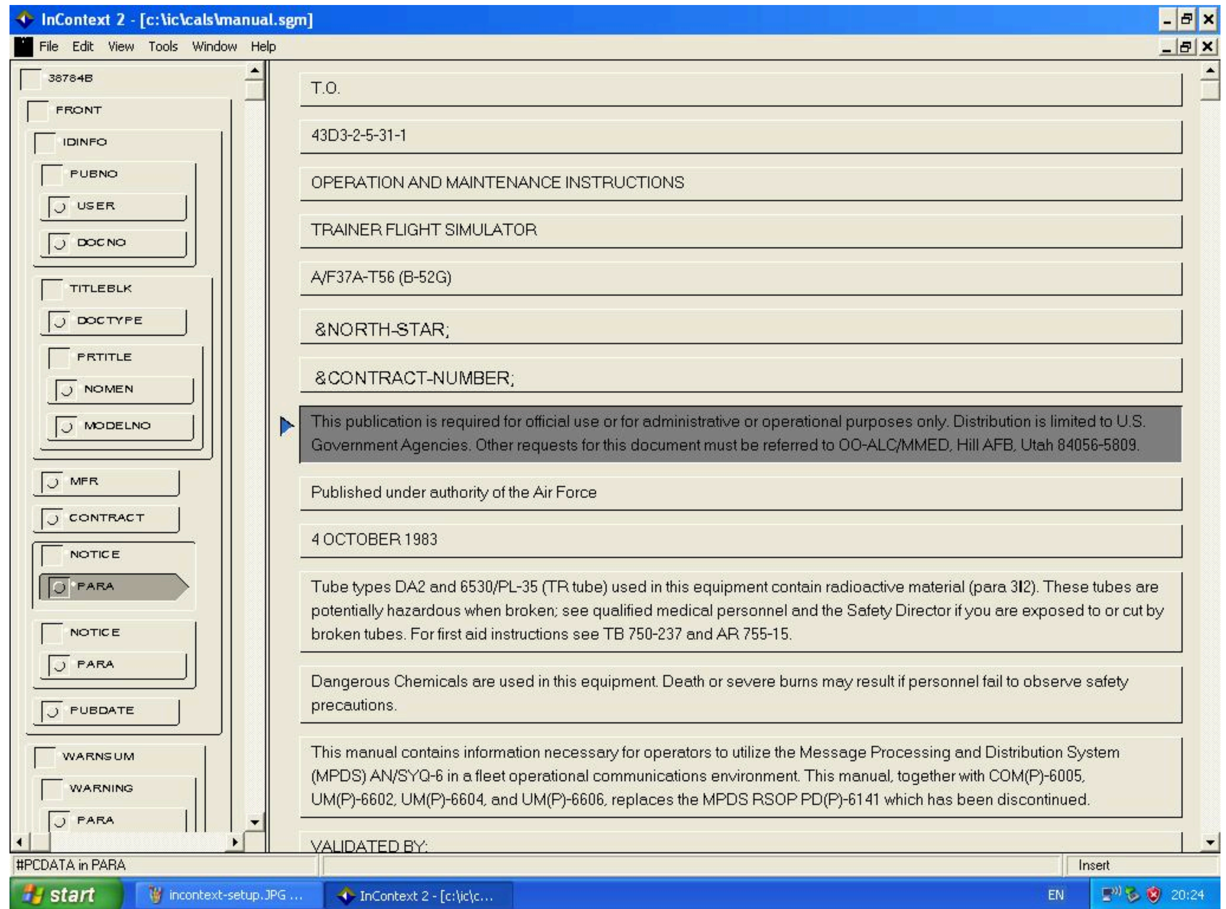


There is a built-in stylesheet system which is reasonably comprehensive except that it does not allow for the formatting of generated content, and attribute values cannot be included in generated content. The result of styling the sample document is also shown on in Figure 5 [19]

2.2.5. InContext

InContext is one of a small number of SGML editors more suited to *data* applications, with the interface using boxes for each element (see Figure 6 [20]). Navigation is provided in a side-panel, using a 3D arrangement of stacked, nested buttons to represent the document tree. This is a good way to display documents where *data* elements occur frequently and text is minimal, but its usability as an editor for writing continuous text is severely limited. As the only available disk was an evaluation copy, it was not possible to open the sample document, only the samples provided by the vendor. Microsoft Office (specifically, Excel, to handle tables) must be installed before installing InContext.

Figure 6. InContext editing a CALS manual



Access to editing in Mixed Content *is* possible, despite the box design: selecting an element type like `para` lets you split the content into nodes, and empty elements are shown as symbols.

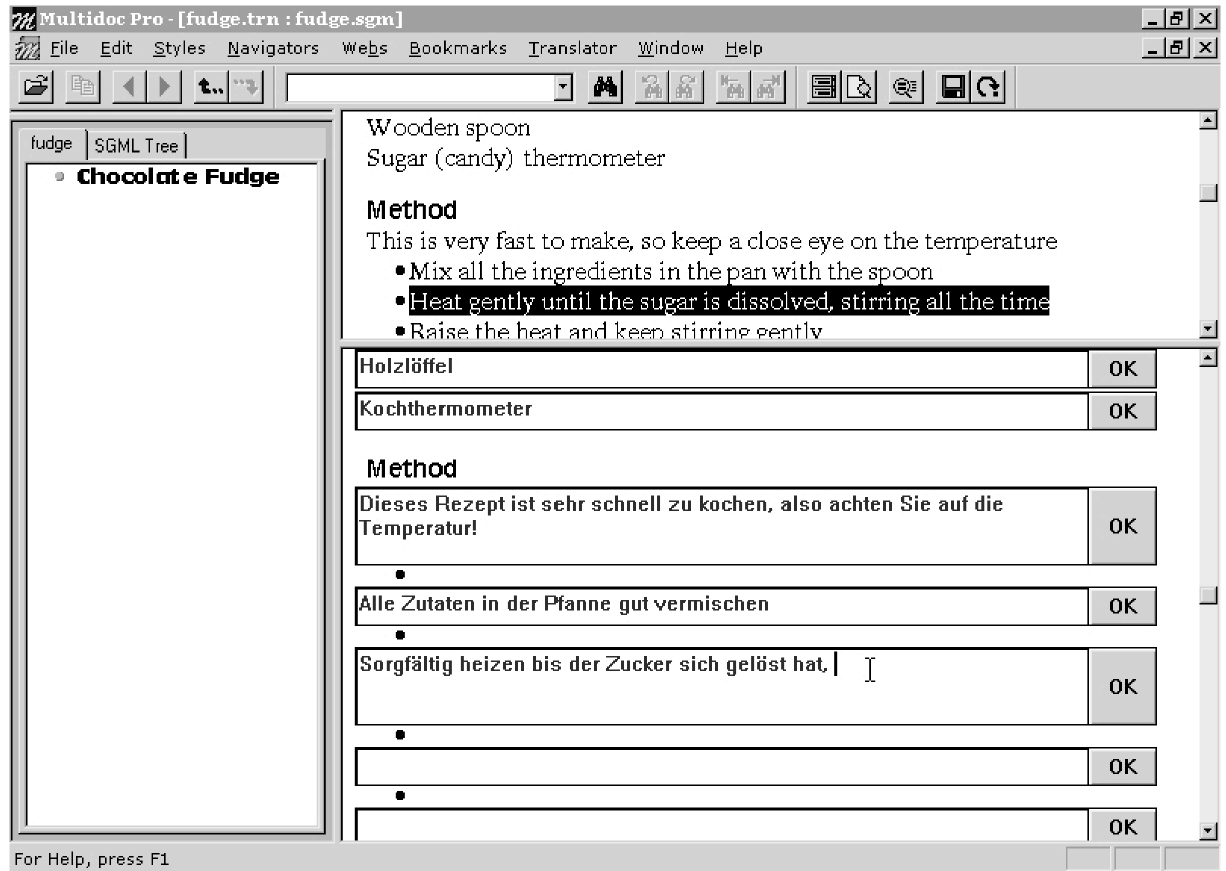
Other software with a related interface includes Microsoft Office InfoPath (2003), and the Citec's MultiDoc Pro Translating Editor

2.2.6. MultiDoc Pro Translating Editor

This is a companion program to the MultiDoc Pro Browser and Publisher, sharing the same interface and the same style files. The objective is to let a translator fill in a parallel document structure with the translated text (*target*) occupying the same element types in element content as the original document (*source*).

Installation is from CD, with a choice of this program, the browser/publisher, and some ancillary programs. Opening the sample recipe document meant using the normalized version (created with sgmlnorm or epcedit) because MDP does not handle files using missing end-tags or attribute minimization.

Figure 7. MultiDoc Pro Translating Editor editing a recipe (original formatted above; translation in boxes below)

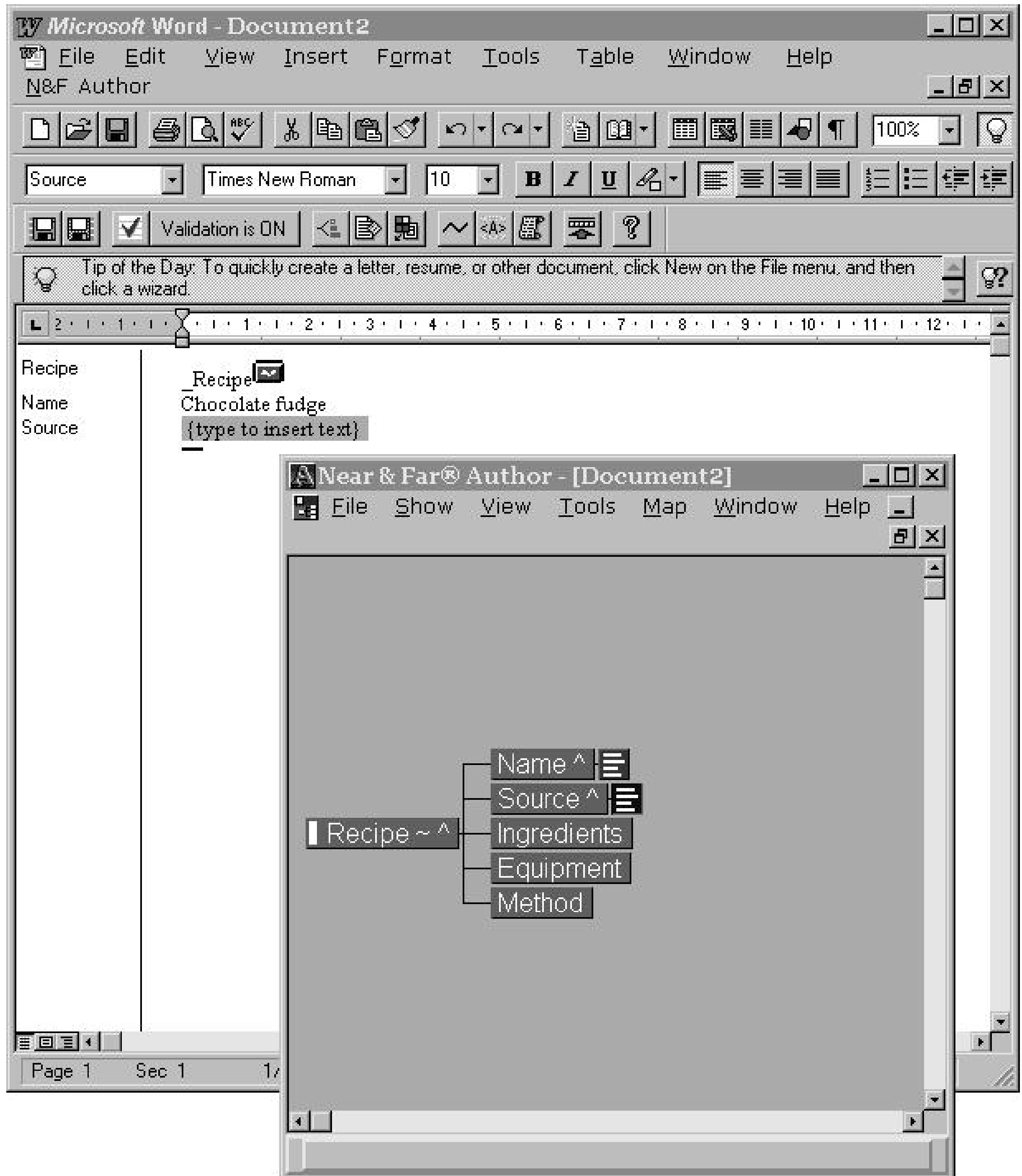


The similarity between the target interface and the one used by InContext is immediately apparent in Figure 7 [21]. The advantage of using their existing browser and stylesheet apparatus to present the source document presumably meant that adding the parallel hierarchy below was unproblematic, except that in Mixed Content, any subelements may be in a different order in the target because different languages have different ways of expressing things.

2.2.7. Near&Far Author for Word

From the same stable as Near&Far Designer (see Section 2.5.1 [30]), this is not a standalone editor but an add-on for Microsoft Word. It adds *Import* and *Create* menu entries for the Word interface which parallel the *Open* and *New* operations. It uses whatever DTD you select, without the need to precompile it, and it uses the same graphical navigation as the companion Designer program (see Figure 8 [22]).

Figure 8. Near&Far Author for Word editing a Word document with named styles



However, as with every other markup tool that interfaces with Microsoft Word, there is a prerequirement that the Word document uses exclusively Named Styles, as this is the only way in which element type names can be bound to recognised spans of text in the document.

As can be seen from the screenshot, the floating window with the document structure from the DTD can be used to guide formation of the document, with the selected element type mapping to a named style, which in turn provides the formatting...which can of course be changed by the author without affecting the markup, although the formatting toolbar is usually disabled to prevent meddling.

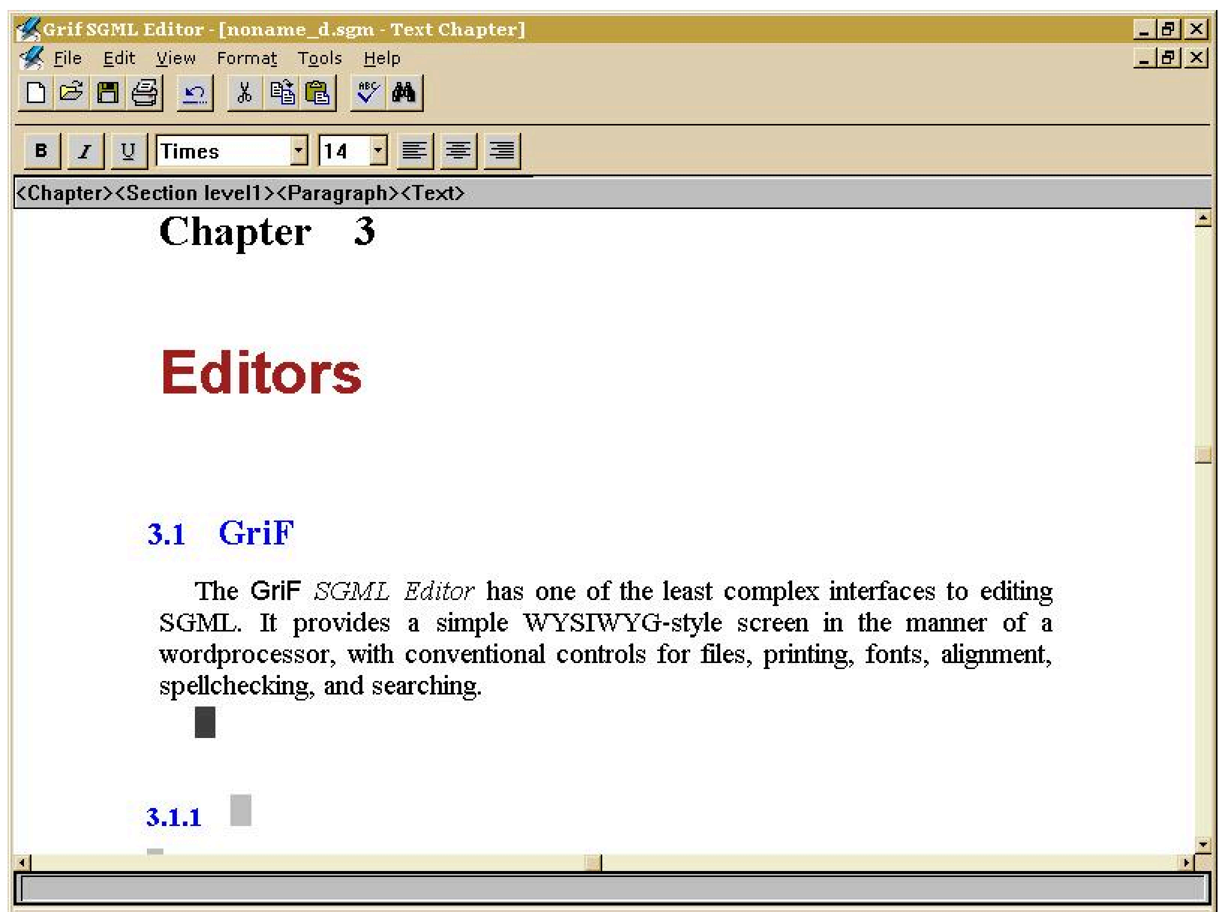
An export function saves the document as SGML. In this way a circular conversion can be obtained, much like Microsoft's SGML Author for Word but with the benefit of executing from within the editor.

2.2.8. GriF SGML Editor

The GriF SGML Editor has one of the least complex interfaces to editing SGML. It provides a synchronous typographic screen in the manner of a wordprocessor, with conventional file management and editing controls. In addition to being popular in business and publishing applications, GriF was the interface used in the Euromath editor.

As with most editors in this class, compiling the DTD and creating a stylesheet are tasks administered separately from the editor, using the Application Builder program. However, (so far as this author is aware) uniquely to GriF, the screen controls still allow the user to impose local (non-element-based) styles on a document. This approach allows almost complete word-processor-style control with independence from the markup: a stylesheet may cause a particular element to default to bold type, for example, but you can override that manually by using a different font, for example, or changing the size, on an entirely independent basis, without affecting the *element* markup in any way. The effect is achieved internally by storing Processing Instructions to record the *ad hoc* styles within the markup, so that these operator-controlled styles are not tied to an element.

Figure 9. GriF's SGML Editor showing text-entry location for the title of section 3.1.1



By default, locations on the screen where character data content is required are identified by gray squares, so text entry for very prescriptive DTDs can be made almost as simple as a form-fill application (many other editors also do this but require scripting).

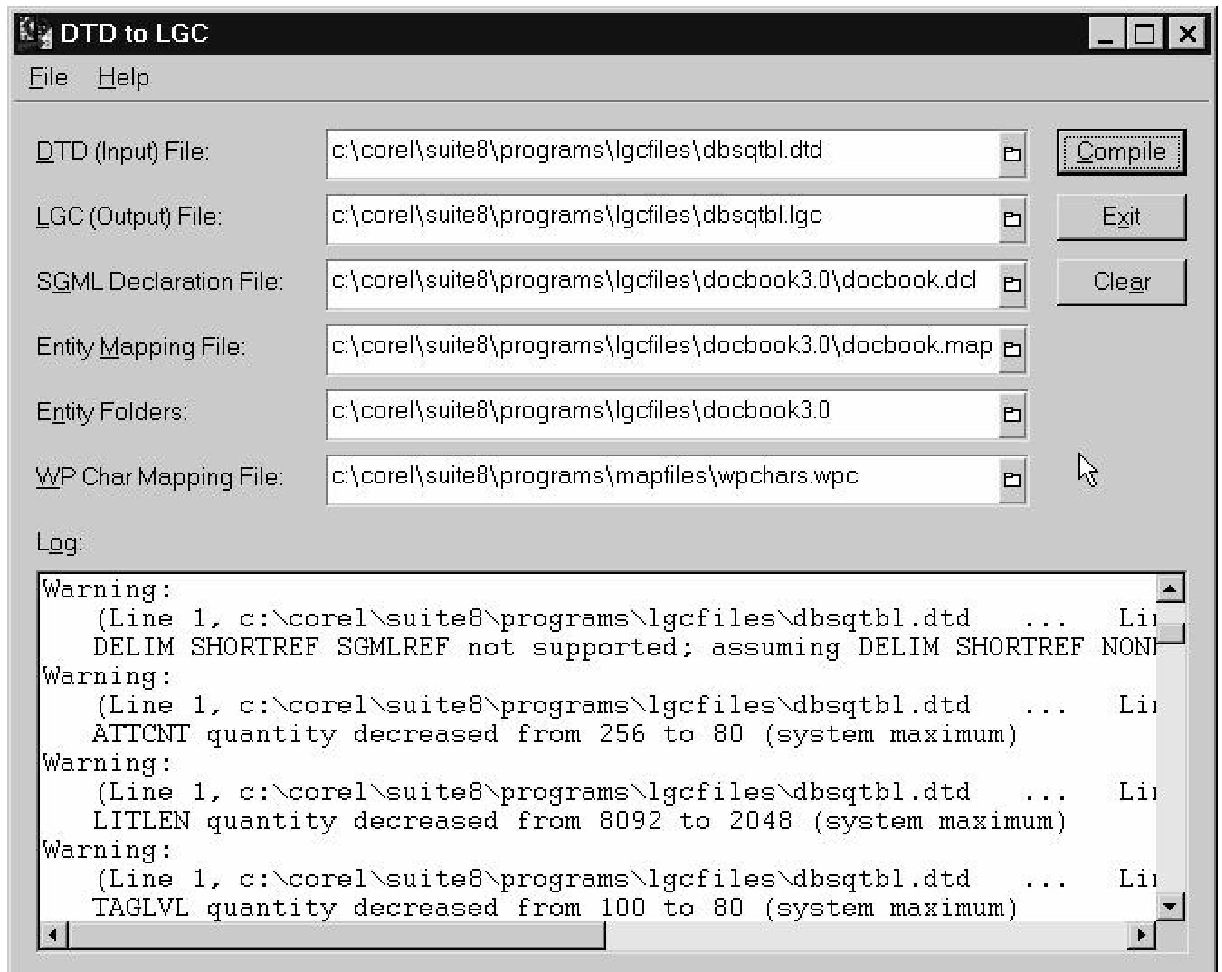
GriF was an early implementer of the feature during text entry that the Enter key performs an element split; that is, it creates a new instance of the current element in element content, if the DTD permits this — for example, pressing Enter at the end of a `para` would create another `para`. If the current content model is at an end (all required elements are present, and no further optional elements are wanted), the TAB key moves to the next location in the document model where input is possible.

2.2.9. WordPerfect+SGML

WordPerfect was for many years *the* dominant wordprocessor, especially in the MS-DOS period. Reputedly even WP's own engineers regarded the character-cell version as the *real* WordPerfect, and superior to the GUI version. However,

embedded inside the Windows version (8) was a real, fully-fledged SGML editor and stylesheet system, providing synchronous typographic editing of SGML documents for the price of a wordprocessor, well below the entry-point for the larger SGML-only editors.

Figure 10. WordPerfect compiling a DTD and creating a stylesheet



Compilation requires choosing the correct character map files for whatever is expressed in the SGML Declaration, but otherwise it is fairly tolerant of the settings, and adjusts them to suit itself. This produces what WP calls a *logic* file, which can then be used to create a stylesheet (template, see Figure 10 [24]) which becomes available along with the document type of the DTD and appears in the wordprocessor's SGML menu (the normal Open/New functions only work for wordprocessor documents).

Figure 11. WordPerfect editing SGML

2.3. Processors

Of the systems in ???TITLE??? [12], Jade is not covered here.

2.3.1. Balise

Balise was known as the Swiss Army Knife of SGML. It can act as a parser and validator, transformation programming language, document manipulator, outliner, pretty-printer, and much else. It consists of a high-level programming language which in effect acts as an API to the document. The interface is the command-line compiler/interpreter.

Unusually (uniquely) it also provides access to the DTD, both logically and syntactically, so that the requirements of the DTD can be queried during document processing; for example finding what other element types are valid at the location of the current one. It also provides access to some non-ESIS parts of the document structure, such as the start and end locations of marked sections.

2.3.2. DAPHNE

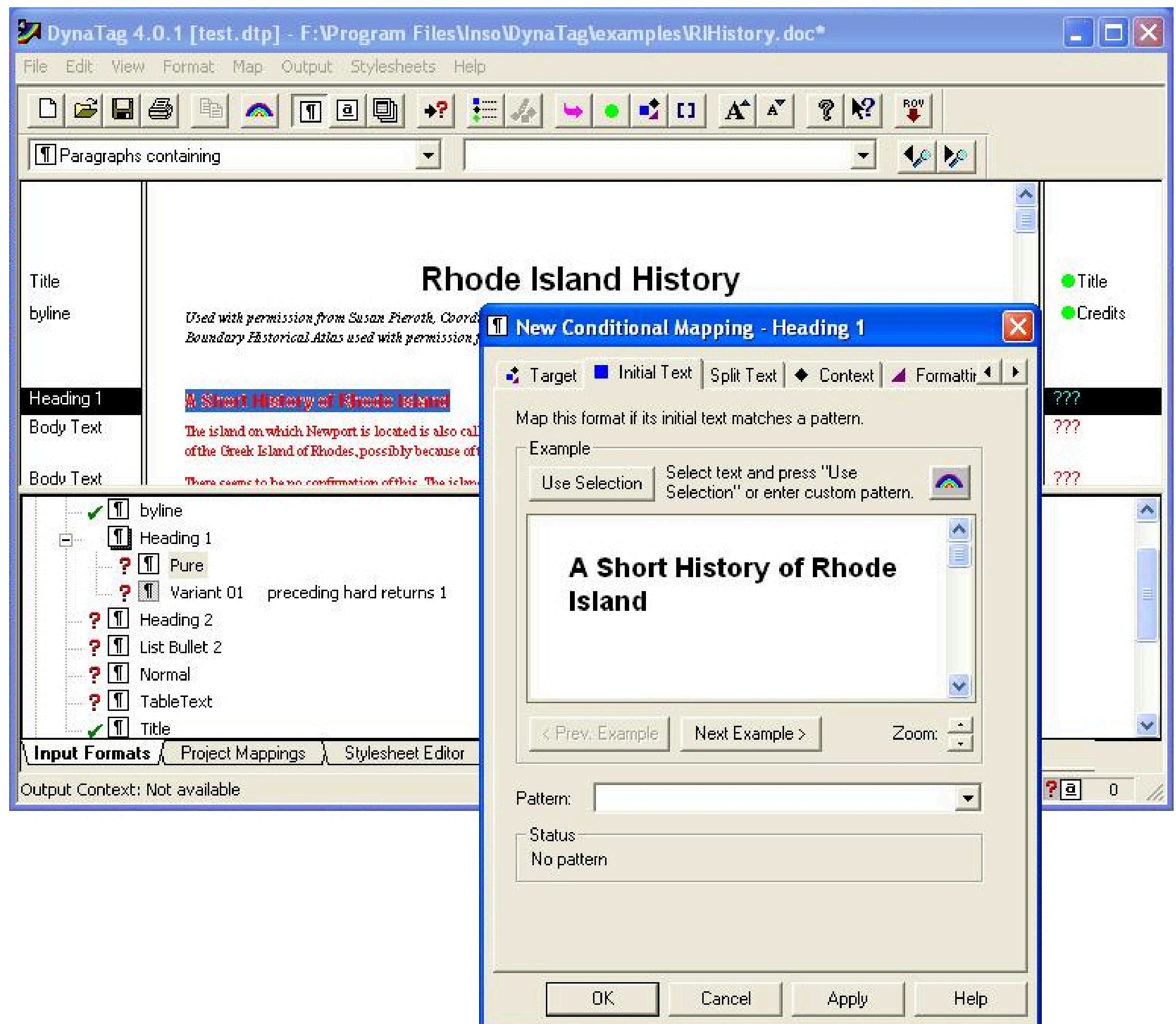
SGML-to-LaTeX converter for VAX/VMS from the German Research Network (DFN) based on the QWERTZ DTD created by the Institute for Applied Information Technology (FIT) at the German National Research Centre for Computer Science (GMD). As far as is known, this is no longer available.

2.3.3. DynaText

There are three components to the DynaText system: DynaTag, which models conversions from Word to SGML; DynaText proper, a system for creating eBooks; and DynaWeb, a web server for dynamic publishing of the eBook documents.

DynaTag displays a Word file with named styles, and graphically lets the operator give a mapping to the desired SGML element, as shown in Figure 12 [26]. Text can be split, combined, omitted, and grouped (providing containment for lists, for example). A common use was to create an intermediate SGML file which would then be transformed by (eg) Omnimark or Balise to the final form. Once the initial mapping was established on one document, additional documents that followed the same style could be added, and the mapping refined. Eventually, the accumulated *knowledge* could be used on entire directories of documents of one pattern for bulk conversion.

Figure 12. DynaTag configuring a Word document for conversion to SGML



DynaText itself could take arbitrarily large SGML documents (perhaps produced by DynaTag), and use an SGML-based stylesheet to render them on-screen as dynamic eBooks, with full-text searching, including Boolean operators and the use of the markup to guide the search. The system was in widespread use in industry and in technical, literary, and academic publishing, and was very influential on later developments (DSSSL, CSS, XSL:FO).

DynaWeb was a HTTP server for Windows NT, performing a similar function to DynaText but serving HTML generated on-the-fly. This meant that fast-changing documents (sourced in Word) could be pushed through a workflow starting with conversion in DynaTag, and once validated, served immediately to the next request.

2.3.4. Omnimark

Until the appearance of XSLT, Omnimark was a frequent choice for SGML conversions. It contained a pattern-matching language which enabled transformation between SGML and non-SGML formats (down-translation), between non-SGML and SGML formats (up-translation), and between arbitrary text formats (cross-translation).

Figure 13. Fragment of Omnimark code showing transformation to LaTeX

```
down-translate

global stream temp

element ABSTRACT
        output "%n\begin{abstract}%n%c%n\end{abstract}"

element ACRONYM
        set buffer temp to "%c"
        output "%g(temp)\index{M}{%g(temp)}"
            when attribute remap isnt specified
        output "\acro{%g(temp)}{%v(remap)}"
            when attribute remap is specified
```

In the example in Figure 13 [27], the `%c` emits the element content (equivalent to XSLT's `apply-templates`); `%n` is a newline; `%g` dereferences a buffer; `%v` dereferences an attribute. A streaming feature meant the document did not have to be read into memory in its entirety; a value not yet encountered in document order (but known to occur) could be referenced, but not dereferenced until the end of processing, by which time the desired value would have been encountered and set as a *referent*).[6]

For a brief period in the late 1990s the product was made available without charge, but this was later abandoned. The software is still available in a much more advanced version for XML and is widely used in publishing workflows.

2.3.5. Microsoft SGML Author for Word

Despite the name, this is *not* an authoring editor. It was (perhaps still is) a plug-in converter from Word to SGML *and back*. In tests conducted for an earlier review, it was able to convert circularly: from Word to SGML, edit the document, convert from SGML back to Word, edit the document, lather, rinse, repeat…losslessly book.

Admittedly it took considerable configuration, but astonishingly, it worked. This was intended to allow a non-SGML person to author a document, have the publications staff convert it and edit it, and then convert back to Word and hand it back to the author to carry on writing or editing, again and again until ready. Microstar's Near&Far Author for Word provides a similar facility, but embedded within the Word interface itself.

2.4. Formatters, including browsers and servers

Of the items in ???TITLE??? [13], 3B2 and Arbortext Publisher are not covered here.

2.4.1. Panorama Publisher and Viewer

The Panorama Free plugin for Netscape was many end-users' first sight of SGML, although if properly set up, they would hardly notice the difference except for the superior formatting. When a Panorama user browsed or followed a link to an SGML document, the plugin (or standalone version) would expect to find two files called (exactly) `catalog` and `entityrc` on the server in the same directory as the SGML file. The `catalog` contained entries resolving the Formal Public Identifier in the document's `DOCTYPE` declaration to a downloadable DTD in the normal way of catalogs:

```
PUBLIC "+//Silmaril//DTD Recipes//EN" "../recipe.dtd"
```

This allows Panorama to download the DTD. The `entityrc` file contains entries matching the FPIs in the `catalog` and providing the names of the stylesheet[s] and navigator[s].

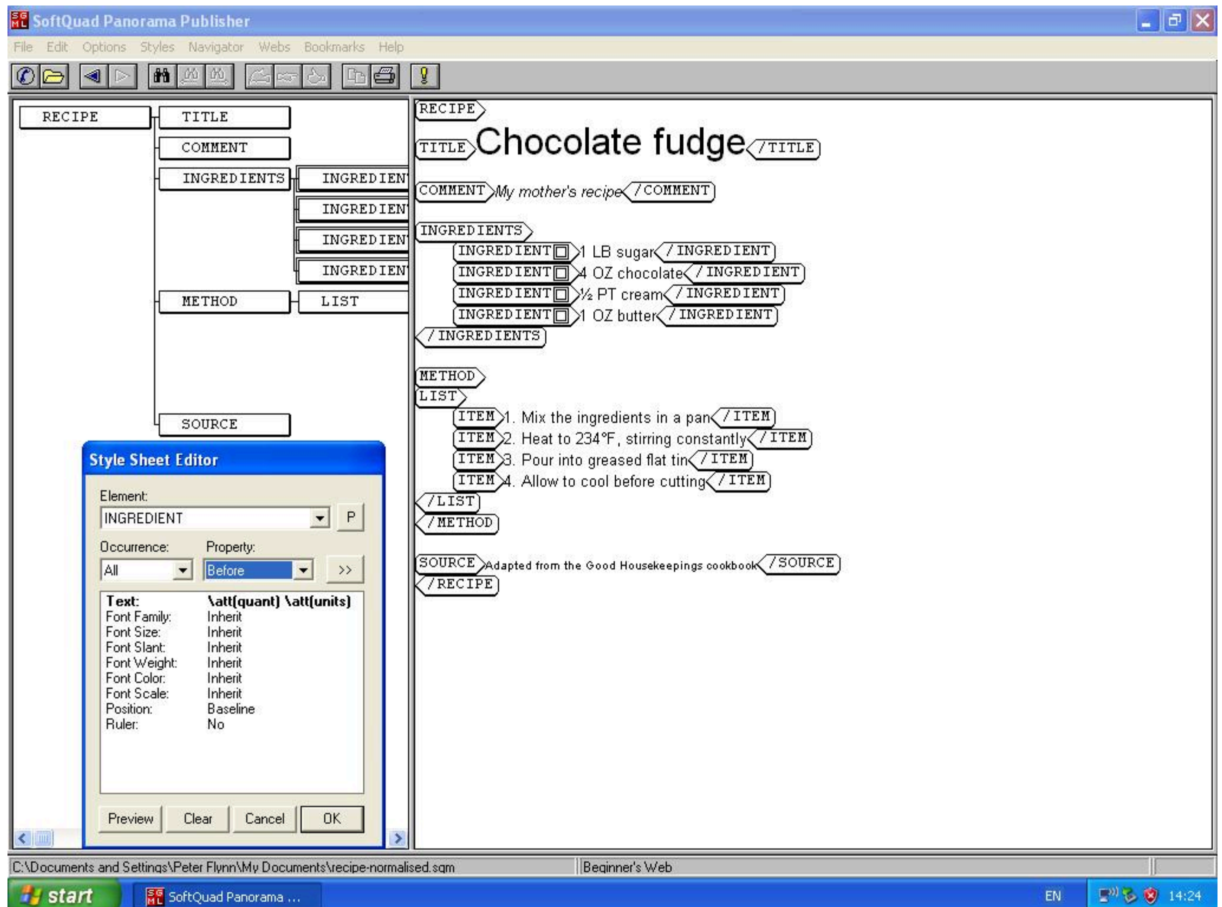
```
PUBLIC "+//Silmaril//DTD Recipes//EN"
  DOCTITLE "recipe,title"
  STYLESPEC "Standard" "recipes.ssh"
  NAVIGATOR "Contents" "recipes.nav"
```

[6]Very large documents with very high numbers of such referents typically caused a brief but audible rattling at the end of processing as the disk drive actuator arm repeatedly sought and wrote the data from whatever temporary location had been created during processing.

Per-document variants of the stylesheets and navigators can be specified in the instance by using Processing Instructions:

```
<?STYLESHEET "NewStyle" "cookbook.ssh">
<?NAVIGATOR "WebVersion" "webrecipes.nav">
```

Figure 14. Panorama Publisher creating styles for a document



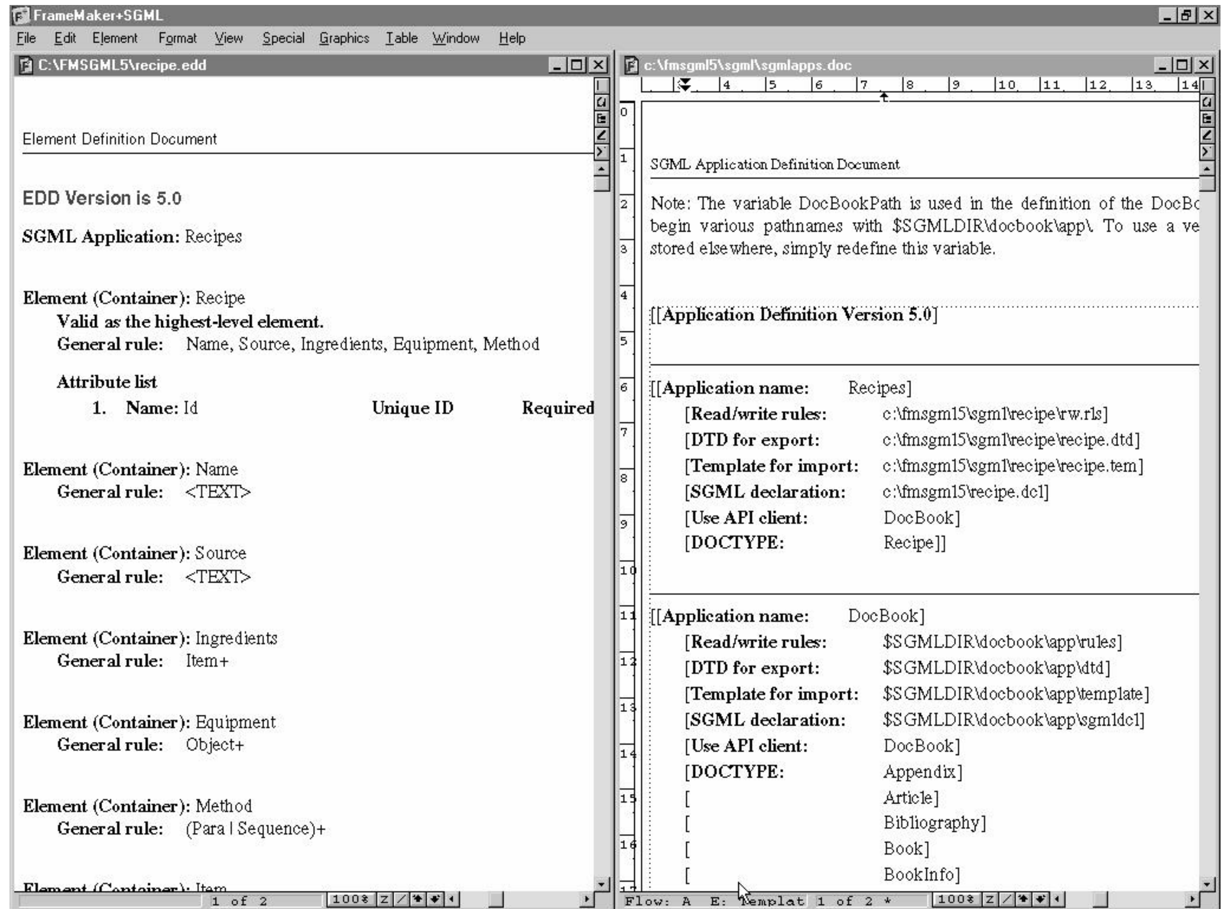
For a large DTD (eg DocBook, TEI, and many industrial schemas) the process of creating the stylesheet is lengthy, the same as it would be for CSS today, but the stylesheet interface is entirely graphical and very easy to use (see Figure 14 [28]).

Opening SGML either locally or over the web offered a significant advantage: the browser obeyed the stylesheet formatting, so you were no longer at the mercy of the web browsers' feeble implementations of CSS; the downside was that in-browser scripting (eg VBScript or Javascript) were not available within Panorama, so it was restricted to classical document-server applications (nonetheless extensive). The biggest advantage, however, was never really taken up: hypertext. Panorama implemented HyTime techreport, so it could handle bidirectional linking, multi-headed (drop-down) links, and — most importantly — you could apply links via the browser without needing write-access to the document because they were stored in your own local file which you could publish on the web, so that other people opening the document could reference your file and see all the links take effect.

2.4.2. FrameMaker+SGML

FrameMaker was another long-time standard for publishing, and had the ability to work with SGML documents since Adobe took it over in the mid 1990s, when it was aimed at the industrial structured-document formatting market. As with other editors, the DTD has to be compiled, in this case to an EDD file, and then a stylesheet created, before any editing or formatting can take place. The early interface for this was forbiddingly complex (see Figure 15 [29]).

Figure 15. FrameMaker+SGML creating styles for a newly-compiled DTD

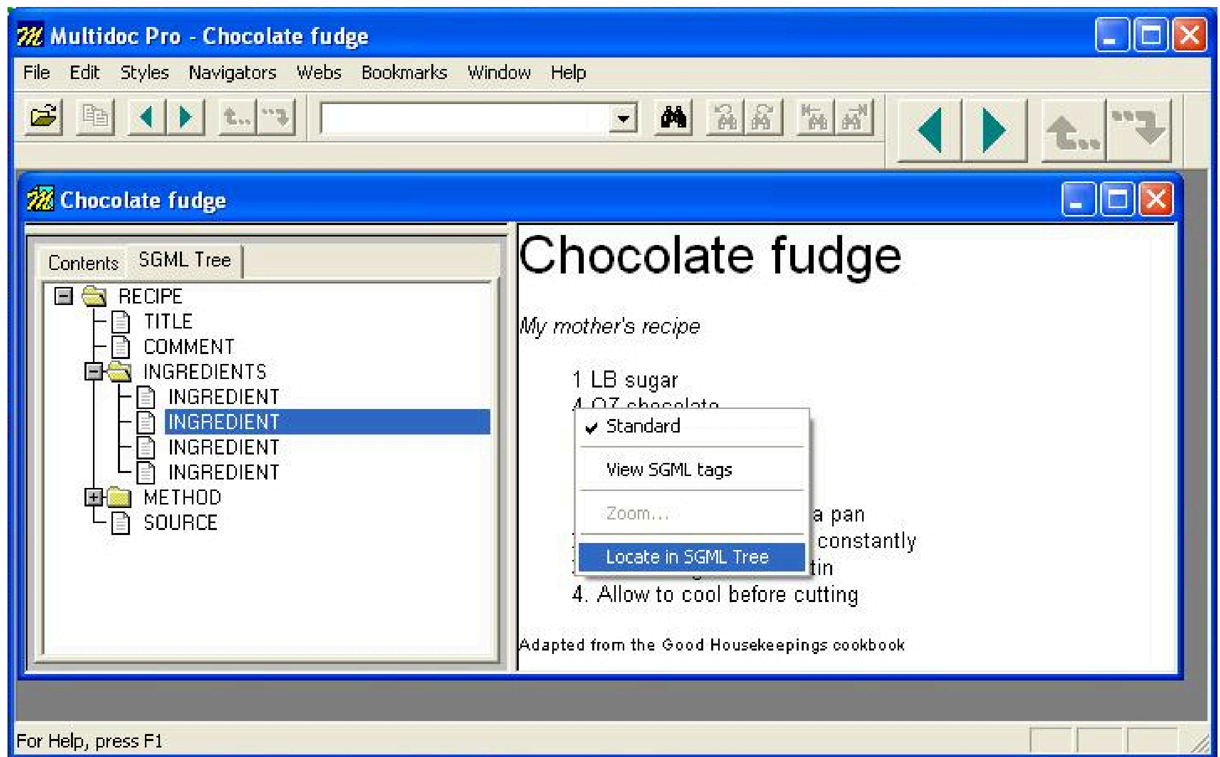


While the system was popular with typesetters, its use with SGML was troublesome for many, with tweaks and adjustments required, and concerns about the way in which SGML was exported, especially as its own (MIF) compatibility export format was very successful. Like ADEPT, it had the ability to apply non-structural *tweaks* after formatting, before pre-press, to adjust visual details not provided for in the code.

2.4.3. MultiDoc Pro Publisher

MDP used the same mechanisms and file structures as Panorama (see Section 2.4.1 [27]), so files prepared for one system could be used in the other.

Figure 16. MultiDoc Pro Publisher viewing a document created in Panorama Publisher



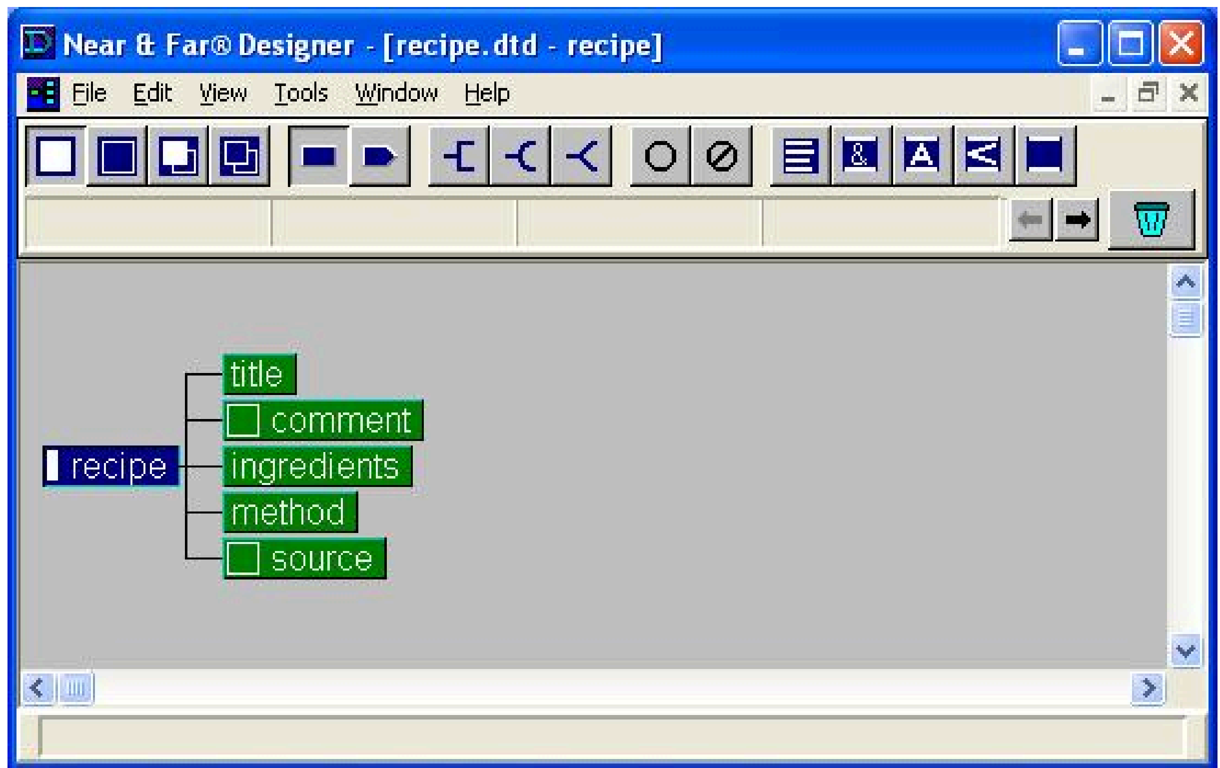
It also implemented the Occurrence Density Display of search results (familiar to modern users as the right-hand bar in a search in some web browsers, giving fine horizontal lines at proportional locations in the height of the window to the length of the document) for extensive search features including the TEI Pointer syntax.

2.5. Other software

2.5.1. Near&Far Designer

A companion to Near&Far Author for Word (see Section 2.2.7 [21]), Designer was a graphical interface to DTD creation and management. It had a simple and effective drag-and-drop paradigm to create element types, add attributes, and establish content models, using symbols to represent the syntax of declarations (eg `EMPTY`, `#REQUIRED`, and the punctuation of content models). Whole chunks of element content could be clicked and dragged around the document model while working to find an optimal way of representing the document.

Figure 17. Near&Far Designer's view of the sample Recipe DTD and the DocBook3 DTD



There were a few drawbacks: importing a DTD meant it had to be *flattened* to a single file — problematic for a heavily modularised DTD like the TEI; it also meant that re-exporting it would lose any conditional parts that had been excluded by the use of Parameter Entities. While it worked well for smaller structures, it was not generally used for industrial or technical DTDs — with the exception that document type designers used it very extensively (and some still do) for the quality of the display, rather than its constructional modelling abilities, partly because the way in which the tree was represented seemed to be recognised by otherwise non-technical clients as immediately comprehensible.

2.5.2. PAT

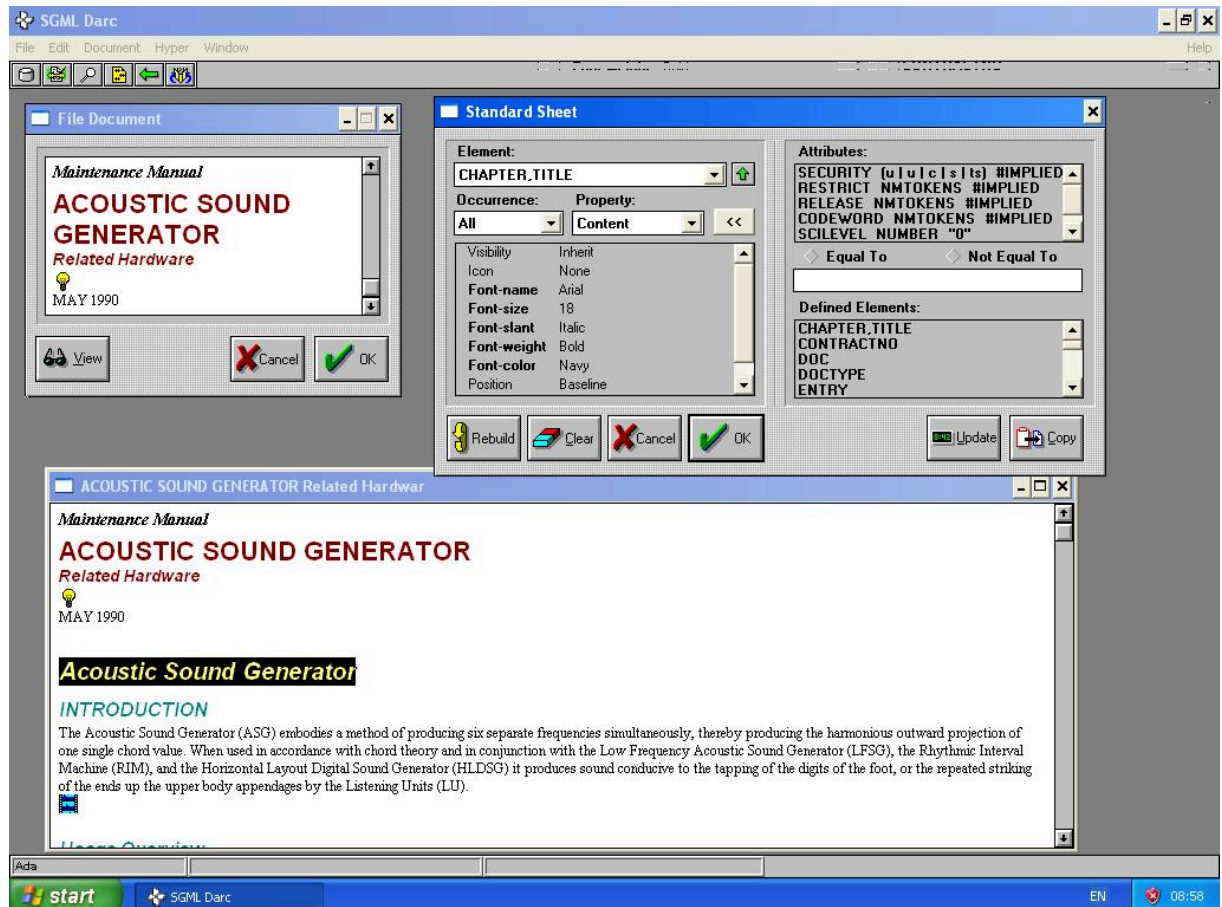
At the time, PAT was claimed as the only native SGML database product incollection. It was created to provide indexing search for the Oxford Dictionaries project at the University of Waterloo, and later commercialised by OpenText Corporation. PAT was available for SunOS 4.1.3 (this author's platform in those days) and was installed for the CELT project for searching their TEI corpus of Irish writing. It continued in use until the platform failed in a lightning strike in 2003 (resuscitation is currently ongoing, as this was the world's ninth web server).

PAT's main advantages were the ease of ingestion of a new corpus (basically a single vast, monolithic SGML document), and the speed of a search. Significant scripting was needed at CELT to turn the KWIC format output into a fully-referenced page for the web, as it meant revisiting each hit to look up the element types needed to compose a reference, and then again to retrieve the reference points themselves, resulting in a slower-than-optimal return, but acceptable in those days.

2.5.3. SGML Darc

The SGML Document Archive from the KTH in Stockholm is perhaps not strictly a native SGML database, but provides some search and extraction facilities. The system was installed from two 3½" floppy disks without problems.

Figure 18. SGML DARC searching a document



The system requires preparation by compiling the DTDs and the stylesheets to be used for the documents to be stored. Ingesting the documents also requires details of which items are to be indexed, and how, so it can be a lengthy task. The indexing, however, is very robust, and retrieval is fast. For a large repository (*archive*) of documents, having them dynamically formatted is a big advantage, because incoming additions to the same DTD would appear completely consistently.

The display engine was later developed by Synex and SoftQuad as Explorer and later Viewport, which in turn informed Softquad's Panorama (viewer and publisher).

3. Conclusions

The structured-document software world has moved on significantly since the virtual replacement of SGML with XML. Some of this is due to improvements in hardware, especially in speed and capacity, and in software capability and compatibility (or at least interoperability), and in language development, particularly Java and Javascript. XML deliberately cut out a lot of facilities from SGML which were underused or added complexity for little gain — the Design Goals of the XML Specification emphasise ease of use and simplicity techreport. The number of people (and companies) using XML is much larger than it ever was for SGML, so there is probably more software available to meet the demand. With better frameworks and raised awareness, vendors, developers, and programmers have generally been paying more attention to usability, so installing and using current software is easier and more reliable than it was in the days of Windows 95/XP and SGML. Modern applications tend to make less fuss, a lesson learned from the so-called Web 2.0 paradigm which emphasises obviousness. So have we learned anything else?

- *Reports of the death of the command line are greatly exaggerated.* People still use it, and its availability in OS X and Windows 10 means that programs originally restricted to UNIX or GNU/Linux are now available on any platform. Most users may not need it, but developers, administrators, and other technical users do, especially for scripted document management functions and for the bulk processing of documents in a workflow.
- *People do still use SGML.* Several consultancies, including the author's, have publishing clients still maintaining SGML systems, for a variety of reasons.

- Lots of the software did still install and execute, which was a surprise. Of those which failed, some were due to faulty media (having been kept for several decades) and some to the OS environment. The clock had to be reset to 1998 for some of the MS-DOS utilities, and there was one unresolved oddity in executing RulesBuilder, which consistently gave a Windows `Divide by zero or Overflow` error.

3.1. Some stuff has been gained.

1. *DTD/Schema resolution has improved.* SGML applications tended to be rather helpless about where to look for the DTD, with each vendor having a different idea of where the right place was. Catalogs fixed most of that problem, but required care and feeding, and Owners were sometimes careless about naming, spelling, and punctuation. XML Catalogs are an improvement, and with the Public Identifiers now less used, most software seems to look in the document folder for the System identifier unless otherwise instructed; or at least it *asks* for it instead of crashing with an error message.

2. *There is more consistency.* The web interface paradigm, which was still in its infancy in 1995, is now the dominant method of interacting with general users — people, even non-computer-users, are expected to know that you click on things to see more — in the same way that office software is expected to work like Word and Excel. Breaking dominant patterns like these needs extraordinary changes and extraordinary benefits, and the new paradigm of everything being clickable, and not necessarily coloured blue and underlined, is an opportunity to ensure that the underlying XML is used for consistency.

3. *The move to using XML* as the storage format for both Open/Libre/Neo Office and Word has been a sea-change in making documents programmatically accessible

4. *The creation of XPath and XSL* (both T and FO) has brought about huge improvements in expressing addressing and programming transformations

5. *The lessons learned in usability* from end-user interfaces like the Panorama/Synex/Citec-type navigation and stylesheet creation windows mean we now have much better facilities for creating in-app navigation and styling tools

6. *There seems to be far less reinvention of the wheel* now, in that many more applications re-use, or build on the shoulders of, existing schemas and DTDs, rather than inventing new ones every time. Writers and speakers have constantly warned about the risks of corporate hubris in writing everything from scratch rather than adopting or adapting a close-match common vocabulary and structure — while at the same time extolling the virtues of modularity and extensibility ???.

3.2. Some stuff has been lost.

1. *Deprecating the Formal Public Identifier* was probably a good move, but using a web URI is nearly as bad. If the GCA had realised what they had in the ISO 9070 Registry, they could have made a big difference. Formal ownership of Names is important.

2. *psgml risks being lost* because of the introduction of nxml-mode, which handles only RNG, and has a very limited control set, making it virtually unusable as a text-document editor

3. Although the *family-tree hierarchical box diagram representation* of the schema or DTD tree is common in many XML editors, none of them yet appears to match the design, clarity, and ease of use of Near&Far Designer

4. *The use of XML in the web browser* has never properly been supported by the browser-makers, for the exact same reasons as the original HTML wasn't. It has to some extent been saved by Saxon/CE and Saxon/JS, but the browsers themselves are a lost cause

5. *Open Source software* (then usually just called *free*) is no better at surviving three decades than commercial software. In fact, commercial software may have the edge, in that it came in boxes, with manuals, CDs, dongles, licence keys, and other stuff, so it got put on a shelf or into a cupboard.

   However, most (but not all) web sites acting as repositories for the *free* software have long since disappeared; but so have almost all of the corporate web sites of the commercial software.

   What is particularly pernicious is that when the owner[s] of a popular and much-used commercial product diversify (or, sadly, die), and it is then sold to another company, the buyers usually knows roughly what they have bought — the first time it happens. But when those buyers are themselves bought, the third owner has less knowledge and interest. By the time it happens again, and maybe again, a once-reputable product is now owned by a manufacturer of children's toys, and has no idea why it also sells a structured-document editor. Let the market decide only works in the textbook economic circumstances of perfect knowledge. In the Real World™ where everything is kept under wraps, nothing is safe.

6. *Hypertext linking* in the HyTime sense never took off. The Panorama-style browsers demonstrated that it was not only possible but easy to use, and anyone who has taught HTML or had to deal with novice designers will know that there is demand for multi-headed and bidirectional links.

A. Sample SGML document

This was file `recipe.sgml`.

```
<!doctype recipe system "recipe.dtd">
<recipe>
  <title>Chocolate fudge</title>
  <comment>My mother's recipe</comment>
  <ingredients>
    <ingredient quant='1' lb>sugar
    <ingredient quant='4' oz>chocolate
    <ingredient quant='&amp;frac12;' pt>cream
    <ingredient quant='1' oz>butter
  </ingredients>
  <method>
    <list>
      <item>Mix the ingredients in a pan
      <item>Heat to 234&amp;deg;F, stirring constantly
      <item>Pour into greased flat tin
      <item>Allow to cool before cutting
    </list>
  </method>
  <source>Adapted from the Good Housekeepings cookbook</source>
</recipe>
```

1. The DTD used in the sample document

This was file `recipe.dtd`.

```
<!ELEMENT recipe - - (title,comment?,ingredients,method,source?)>
<!ELEMENT title - - (#PCDATA)>
<!ELEMENT comment - - (#PCDATA)>
<!ELEMENT source - - (#PCDATA)>
<!ELEMENT ingredients - - (ingredient+)>
<!ELEMENT ingredient - o (#PCDATA)>
<!ATTLIST ingredient quant CDATA #REQUIRED
                     units (g|Kg|dl|l|oz|lb|pt|cup|0) #REQUIRED>
<!ELEMENT method - - (para|list)>
<!ELEMENT para - - (#PCDATA)>
<!ELEMENT list - - (item+)>
<!ELEMENT item - o (#PCDATA)>
<!ENTITY deg CDATA "&#176;">
<!ENTITY frac12 CDATA "&#189;">
```

2. The SGML Declaration used for the sample document

This was file `sgml.dec`.

```
<!SGML  "ISO 8879:1986"
--
        Document Type Definition for the HyperText Markup Language
        as used by the World Wide Web application (HTML DTD).
```

```
        NOTE: This is a definition of HTML with respect to
        SGML, and assumes an understanding of SGML terms.

        If you find bugs in this DTD or find it does not compile
        under some circumstances please mail www-bug@info.cern.ch
--

CHARSET
         BASESET  "ISO 646:1983//CHARSET
                   International Reference Version (IRV)//ESC 2/5 4/0"
         DESCSET  0   9   UNUSED
                  9   2   9
                  11  2   UNUSED
                  13  1   13
                  14  18  UNUSED
                  32  95  32
                  127 1   UNUSED
     BASESET   "ISO Registration Number 100//CHARSET
                ECMA-94 Right Part of Latin Alphabet Nr. 1//ESC 2/13 4/1"
     DESCSET   128 32 UNUSED
               160 95 32
               255  1 UNUSED


CAPACITY        SGMLREF
                TOTALCAP        150000
                GRPCAP          150000

SCOPE    DOCUMENT
SYNTAX
         SHUNCHAR CONTROLS 0 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 31 127 255
         BASESET  "ISO 646:1983//CHARSET
                   International Reference Version (IRV)//ESC 2/5 4/0"
         DESCSET  0 128 0
         FUNCTION RE          13
                  RS          10
                  SPACE       32
                  TAB SEPCHAR  9
         NAMING   LCNMSTRT ""
                  UCNMSTRT ""
                  LCNMCHAR ".-"
                  UCNMCHAR ".-"
                  NAMECASE GENERAL YES
                           ENTITY  NO
         DELIM    GENERAL  SGMLREF
                  SHORTREF SGMLREF
         NAMES    SGMLREF
         QUANTITY SGMLREF
                  NAMELEN  34
                  TAGLVL   100
                  LITLEN   1024
                  GRPGTCNT 150
                  GRPCNT   64

FEATURES
  MINIMIZE
    DATATAG  NO
    OMITTAG  YES
    RANK     NO
    SHORTTAG YES
  LINK
```

```
    SIMPLE   NO
    IMPLICIT NO
    EXPLICIT NO
  OTHER
    CONCUR   NO
    SUBDOC   NO
    FORMAL   YES
  APPINFO    NONE
>
```

B. Software and documentation available

This is a list of all the material accessible. It is being made available to those who are prepared to act as custodians of what — in some cases — may be the world's last executing or readable copy.

- Advent 3B2 SGML v.2 manual only
- Arbortext ADEPT and Document Architect, 5.4.1 3½" diskettes and docs, trial licence
- SoftQuad Author/Editor 3.5 with RulesBuilder (DTD compiler), manuals and CDs
- AIS Software Balise 3.1 Reference Manual, Installation Note (2), Tutorial (2), Programmer's Guide, 3×3½" diskettes, parallel-port dongle (required)
- DFN DAPHNE User Manual 3.0 (in German) Deutsches Forschungsnetz Report 51 (April 1988).
- KTH SGML DARC on 2×3½" diskettes, experimental version with manual
- Arbortext EPIC, boxed, CDs and docs, CPUID licence
- EBT DynaText, DynaTag, and DynaWeb Document Preparation, Features, Introduction, Publisher's Guide, Customising, Server, Online Publishing Guide, Publishing Setup, InSted Users Guide; CDs
- EMT EuroMath Users Guide v.2
- Adobe FrameMaker+SGML 5.1.1, boxed, manuals, evaluation copy. This list is alphabetical by product name.
- GriF manuals and 3½" diskettes
- InContext InContext 3½" installation evaluation disk only, no licence key
- Citec MultiDoc Pro Publisher 2.5
- Microstar Near&Far Author 2.0 on 3½" diskettes (needs Word 6 or 7 *only*)
- Microstar Near&Far Designer on 3½" diskettes for Windows 3.1 or 95/XP
- Omnimark MS-DOS V2R5 manuals and 3½" diskettes
- SoftQuad Panorama Viewer and Publisher manuals, 3½" diskettes and CD
- PAT 3.3 Users Guide (Heather Fawcett) New Oxford English Dictionary Centre, University of Waterloo, Canada
- PAT 3.4 Release Notes (OpenText Corporation)
- PAT Workstation Guide (Heather Fawcett) New Oxford English Dictionary Centre, University of Waterloo, Canada
- Quicksoft PC-Write 3.02 manual and 5¼" diskettes
- Microsoft SGML Author for Word documentation and 3½" diskettes
- OUP SGML Tagger documentation and 3½" diskette
- WordPerfect 8 with SGML, box only, but includes software on the CD from book

The following are books or other documents, not documentation, but the book has a CD with a collection of *free* software.

- book
- book
- GCA SGML '91 Conference Proceedings Providence, RI
- GCA SGML '95 Conference Proceedings Boston, MA
- GCA SGML '96 Conference Proceedings Boston, MA
- GCA SGML/XML '97 Conference Proceedings Washington, DC
- GCA SGML/XML Europe '98 Conference Proceedings Paris, France
- Mulberry The SGML Hornbook, paper, 8pp.

Some CDs are just conference papers, others are mixed software

- SGML '97 Conference
- SGML/XML '98 Conference
- XML '99 Conference
- XML 2003 Conference
- Extreme Markup 2002
- Markup Technologies '99
- SGML '96 Power Tools
- SGML '97 Power Tools
- SGML '97 Power Tools
- XML '99 Power Tools

Bibliography

[book] *Text Processing and Typesetting with Unix*. David Barron and Mike Rees. 1987. Addison-Wesley. Reading, MA. 447. 0201142198.

[techreport] ISO. *Information technology -- Hypermedia/Time-based Structuring Language (HyTime)*. ISO 10744:1992. International Organization for Standardization. Geneva. ISO 10744. 1992.

[article] "The Implementation of the Amsterdam SGML Parser". Jos Warmer and Sylvia Van Egmond. cajun.cs.nott.ac.uk/compsci/epo/papers/volume2/issue2/epjxw022.pdf. *Electronic Publishing*. July 1989. 2. 2. 65–90. 0894-3982.

[techreport] ISO. *Standard Generalized Markup Language*. ISO 8879:1985. International Organization for Standardization. Geneva. ISO 8879. 1985.

[book] Peter Flynn. *Understanding SGML and XML Tools*. SGML Toolbook. Kluwer. Boston. 0792381696. May 1985.

[incollection] "Open Text Corp". Seybold, Inc. *DBMS Support of SGML Files*. 3 October 1996. Bob DuCharme. snee.com. http://www.snee.com/bob/sgmldbms.html. .

[techreport] Tim Bray, Jean Paoli, and Michael Sperberg-McQueen. *Extensible Markup Language Version 1.0*. XML. World Wide Web Consortium. Cambridge, MA. 10 February 1998. 2. REC-xml-19980210. https://www.w3.org/TR/1998/REC-xml-19980210.

[book] Eve Maler and Jeanne el Andaloussi. *Developing SGML DTDs*. from Text to Model to Markup. Developing DTDs. Prentice-Hall. Upper Saddle River, NJ. 1999. 0-13-309881-8.

[article] "More About Custom DTDs". W3C. https://alistapart.com/article/customdtds2/. *A List Apart*. 17 May 2005. 1534-0295.

[inproceedings] "Your Standard Average Document Grammar". Not just not your average standard. Peter Flynn. https://www.balisage.net/Proceedings/vol19/html/Flynn01/BalisageVol19-Flynn01.html. *Balisage*. Balisage 2017. 2017. Rockville, MD. . Balisage Series om Markup Technologies. Rockville, MD. 1947-2609.

[book] Charles Goldfarb, Steve Pepper, and Chet Ensign. *SGML Buyer's Guide*. Prentice Hall PTR. Upper Saddle River, NJ. 1988. 0136815111.

xprocedit, A Browser-Based Open-Source XProc Editor

Marco Geue, Hochschule Merseburg

Gerrit Imsieke, le-tex publishing services GmbH

Abstract

A visual XProc editor can serve at least two purposes: Communicating the process flow to non-programmers and easing the notoriously steep learning curve for programmers.

An implementation using the Javascript framework JointJS and the in-browser XSLT 3 processor Saxon-JS is demonstrated, along with the challenges of supporting XProc's peculiarities in a generic graph editing framework.

1. Introduction

Visual programming languages have been around since the 1960s (see, for example, [Boshernitsan1998]). Visual programming often uses the dataflow programming paradigm, whose description XProc matches exemplarily:

> Applications are represented as a set of nodes (also called blocks) with input and/or output ports in them. These nodes can either be sources, sinks or processing blocks to the information flowing in the system. Nodes are connected by directed edges that define the flow of information between them. [BoldtSousa2012]

XProc is an "XML Pipeline Language" that was first specified in 2010 [XProc1]. Its main purpose is the orchestration of XML validation and transformation tasks that traditionally is done with scripting languages and tools such as Ant, Make, shell scripts, or custom Java programs. Two advantages of XProc are: 1. Processing happens in main memory, which means documents need not be serialized or even stored to disk between processing steps, and there are no penalties in JVM startup times that are often associated with shell script or Makefile orchestration. 2. There is no global state in an XProc pipeline, a thing that is responsible for unexpected side effects in Ant, Make or Shell scripts and that makes these programming languages unsuited for encapsulating possibly complex, re-usable functionality. XProc provides this encapsulation and composability in its processing blocks, the so-called "steps."

It has been argued that, given the advances in XPath 3.1 and XSLT 3.0, together with extension modules for dealing with binary data, archives, HTTP APIs, etc., XSLT itself can now be used as a replacement for XProc pipelines [Quin2019]. While this is possible, XSLT does not offer or enforce the degree of processing block encapsulation that XProc offers and that is a prerequisite for creating a visual editor that lets users assemble predefined building blocks.

While XProc version 1.0 is primarily focused on processing XML documents, XProc 3.0 [XProc3] knows other document types, namely text, HTML, and JSON, but also arbitrary binary files, as first-class citizens. XProc's main applications so far are in publishing, and publishing these days requires handling of non-XML HTML and JSON, in particular. XProc 3.0's embrace of XPath 3.1 as its expression language and the XSLT and XQuery serialization 3.1 specification [Serialization31] are key enablers of its more "webby" capabilities.

XProc 3 programs continue to be written in XML syntax, at least this is the only serialization that is specified for 3.0. When writing a browser-based editor, there probably needs to be some translation between the editor's representation of a pipeline, be it HTML, SVG, or JSON, and XProc's XML syntax. As it will be shown, the solution presented in this paper uses JSON data structures as the internal representation. Translation from XProc XML to this internal model and back to XProc XML will be done with XSLT 3.0 in the browser and the XPath 3.1 functions that convert between JSON and XML.

While the concepts of XProc seem straightforward at first glance – processing steps whose outputs connect with inputs of other steps – many users have reported difficulties in writing actual pipelines. This is partly due to the syntax. The concept of primary inputs and outputs and default readable ports (that is supposed to make pipeline documents less verbose) contributes to this reportedly steep learning curve.

Having experienced graph editors that also know the concept of multiple ports per processing unit, namely the Blender node editor [Blender] and the Lego Mindstorms visual programming environment, we at le-tex thought that connecting existing XProc steps graphically will help overcome the initial learning difficulties. It will also be useful to visualize complex XProc pipelines since the XML representation offers almost no visual clues (apart from adjacent primary output/input ports) which steps are connected and how data flows through the pipeline.

2. Why is XProc Special?

XProc pipelines are basically directed acyclic graphs where the processing steps are the nodes and the input/output connections are the edges. However, when we at le-tex first tried to use generic browser-based graph editing frameworks for XProc in 2014, we discovered that some fundamental XProc properties are not well supported, to wit: Multiple docking ports per node, the distinction between input ports and options, the distinction between parameter and document inputs, encapsulation/sub-graphs, and default readable ports.

Encapsulation, for example, is a powerful feature of XProc that allows to expose a potentially complex pipeline as an apparently monolithic building block with a well-defined interface and opaque innards. Some graph editing frameworks are able to fold a sub-graph so that it occupies less screen real estate. This is no genuine encapsulation though since it requires folded sub-graphs to be copied and pasted rather than re-used. It does not allow the "write once, use many times" approach that XProc's language design supports so well.

This kind of encapsulation can be seen in other functional languages, too, and there are graphical editors for other programming languages. What sets XProc apart is the ability of a processing step to produce many different outputs that

don't need to be consumed at once (or at all). This is useful for example when an encapsulated multi-step conversion pipeline produces the conversion result on one port and intermediate results and validation reports for the input and output on other ports. But this multi-valued, non-simultaneously consumed outputs deviate sufficiently enough from common programming paradigms as to render visual editors for these languages unsuited for editing XProc.

Another peculiarity, XProc's concept of "primary" and "default readable ports" is meant to make pipeline authoring less verbose: The primary port of adjacent (in document order) steps connect implicitly, without the need to establish explicit connections. On the two-dimensional canvas of a visual pipeline editor, however, there is no canonical document order. Turning the 2-D representation into a linear XProc XML document, a task that the visual XProc editor needs to perform, will become an optimization problem where a score needs to be attached to multiple possible serializations, rewarding implicit connections via default readable ports. Alternatively, users of the graphical editor could be forced to make XML document order explicit, a thing that we wanted to avoid for usability reasons.

This means that although XProc seems to be a perfect candidate for visual or dataflow programming, its reliance on XML serialization is an extra challenge when converting the natural graph to an XML representation that actually helps people writing their first pipelines.

One should acknowledge that a visual XProc programming environment will probably never replace actual coding. To a large extent this is due to the amount of XSLT that many pipelines orchestrate. At least this is what our experience as developers of the transpect framework [transpect] tells us. In most of our pipelines and libraries, the core tasks will be performed by XSLT stylesheets. This is not a shortcoming of XProc but rather a feature. Apart from ripping apart XSLT micropipelines, we don't strive at replacing what we do in XSLT with XProc steps. We do see a huge benefit in continuing to use XSLT's template matching and import mechanisms while encapsulating multi-step XSLT, zip/unzip, HTTP request, validation, etc. pipelines in well-defined XProc step signatures with possibly multiple outputs.

3. Selecting a Graph Editing Framework

The criteria for selecting a graph editing framework were partly XProc-related, partly informed by current software development trends and user expectations:

- browser-based
- interactive
- customizable
- provides ports
- supports encapsulation (hierarchical graphs)
- open source

A market research of graph editing frameworks, excluding those that don't run in browsers, led to the following score matrix:

	GoJS	JointJS	NoFlo/FloHub	vis.js	yFiles
interactive	yes	yes	no/yes	yes	yes
customizable	yes	yes	yes	yes	yes
encapsulation	no	yes	yes	no	no
ports	no	yes	yes	no	yes
open source	no	yes	yes/partly	yes	no

The Javascript framework that was selected for xprocedit is JointJS [JointJS]. Although it needs to be said that XProc pipelines are, by far, not a graph type that is supported by JointJS out of the box, its basic support for graph editing, for storing a graph model and for rendering a model as SVG has been very helpful.

NoFlo was a contender, but JointJS was chosen after initial experiments were promising. NoFlo didn't enter the experimental phase, therefore we cannot say whether it would have required less customizing.

Auto-layout was not an initial requirement, but many of the frameworks support it and it will be put to use in particular after loading an existing XProc pipeline.

The choice of XSLT 3 and Saxon-JS for converting back and forth between the browser representation (be it SVG or Javascript/JSON) and XProc XML was never really disputed. This is in spite of Saxon-JS not meeting the open-source requirement that we imposed on the graph editing framework. The rationale for this is that we regard XSLT 3 in the browser as a strategic choice of technology, and we want to support the only existing implementation. Graph libraries for the Web, on the other hand, do too little for XProc-style graphs out of the box to justify an investment in a commercial graph library.

4. Solution

The solution, called xprocedit [xprocedit], consists of an extension to the JointJS shape model in order to accommodate different kinds of XProc steps (atomic, compound, multicontainer; standard/user-defined), primary/non-primary ports, and options that act like input ports but are displayed differently and can have default values.

More details are given in one of the author's master's thesis [GeueMT].

XProc 3.0 is currently in the later stages of being specified. One XProc 1.0 concept that 3.0 does away with is parameter ports. It was decided that parameter ports will not be supported by xprocedit. However, for pipelines that don't use parameter ports, an XProc 1.0 serialization is still available. This is because the generated pipelines need to be tested but XProc 3.0 processors are not widespread yet. Support for XDM 3.1 maps, which serve as a replacement for parameter ports when supplied as a parameter option, is currently being implemented in xprocedit. It should be possible, if users are interested and if funding is available, to use xprocedit as an XProc 1.0→3.0 migration tool, converting parameter ports to map options during import.

An issue where an interactive editor may help is validation, or rather, forcing the user to only create valid pipelines. Ideally, a graphical pipeline editor will not, for example, let users connect two steps with a directed edge that points from input to output, from input to input, or from a step to itself. Another check that is built into the tooling might prevent users from connecting an output port that may emit multiple documents to an input port that only accepts a single document. To that end, a certain amount of custom Javascript application logic had to be employed, and this is not finished yet. Verifying that the declared content types of an output port match the content types that an input port accepts is another future custom Javascript validation that is interesting in designing XProc 3.0 pipelines with their support for non-XML documents.

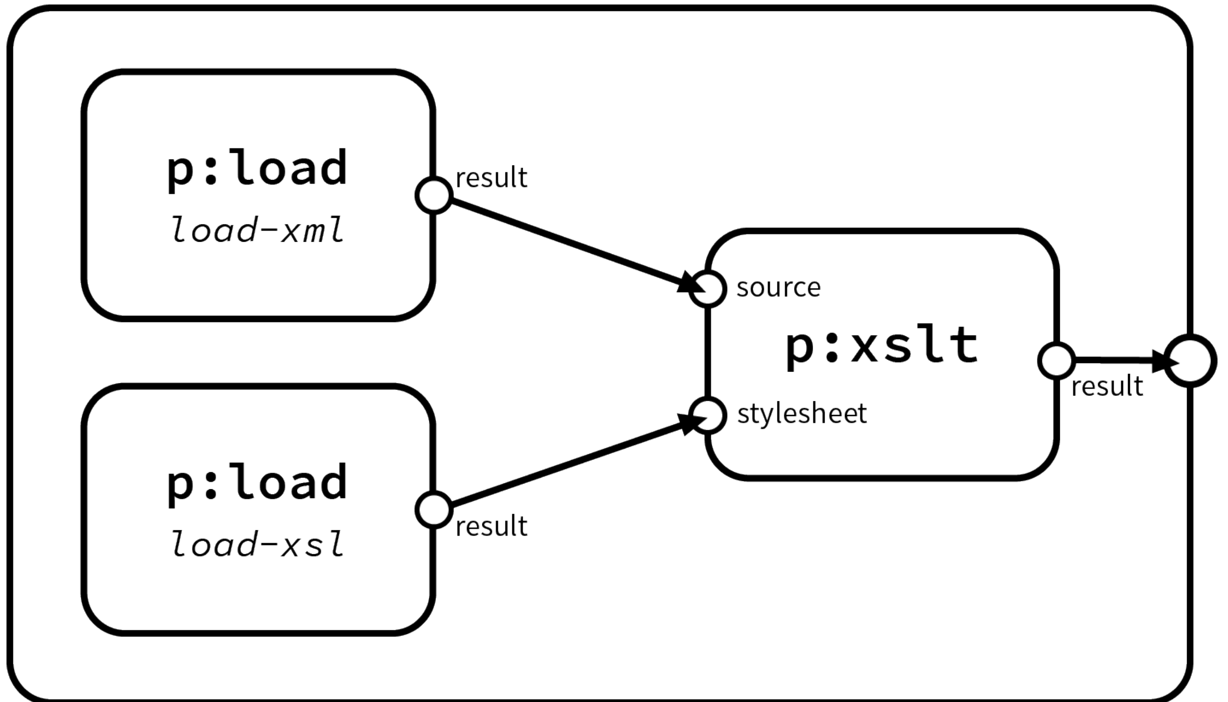
Some constraints, such as whether there are no loops in the pipeline or whether all required inputs are connected, are currently only checked upon export, via XSLT, and with poor error reporting.

As mentioned in the previous section, export from the internal Javascript model via its JSON serialization is performed by XSLT 3.0 transformations in the browser, starting from the result of applying `fn:json-to-xml()` to the JSON model.

The framework maintains an internal Javascript object representation of the graph. It was decided that the XProc XML document be generated from the JSON serialization of this model, rather than from the SVG rendering. The main reason is that, although the JSON representation also contains some layout information, it is considered as more stable than an SVG rendering. The tool that was chosen for generating the XProc XML document, Saxon-JS, supports XSLT 3 and XPath 3.1, and therefore it is equally capable of transforming JSON documents to XML as it is capable of transforming SVG to another XML vocabulary. What finally tipped the scale in favour of converting JSON rather than SVG to XProc was symmetry: There also needs to be an import process that imports pipelines and step libraries to the internal model, which is Javascript/JSON rather than SVG.

An important aspect when generating pipelines for future human editing in XML format is this: There can be multiple equivalent XML representations of a given graph. Consider the following pipeline (Figure 1 [43], taken from [GeueMT]):

Figure 1. A Simple XSLT Pipeline



When serializing this pipeline as an XProc XML document, two variants are possible:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:c="http://www.w3.org/ns/xproc-step" version="1.0">
  <p:output port="result" primary="true"/>

  <p:load name="load-xsl" href="test.xsl"/>

  <p:load name="load-xml" href="test.xml"/>

  <p:xslt>
    <p:input port="stylesheet">
      <p:pipe port="result" step="load-xsl"/>
    </p:input>
    <p:input port="parameters"><p:empty/></p:input>
  </p:xslt>

</p:declare-step>
```

and

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:c="http://www.w3.org/ns/xproc-step" version="1.0">
  <p:output port="result" primary="true"/>

  <p:load name="load-xml" href="test.xml"/>

  <p:load name="load-xsl" href="test.xsl"/>

  <p:sink/>

  <p:xslt>
    <p:input port="source">
      <p:pipe port="result" step="load-xml"/>
    </p:input>
```

```
    <p:input port="stylesheet">
      <p:pipe port="result" step="load-xsl"/>
    </p:input>
    <p:input port="parameters"><p:empty/></p:input>
  </p:xslt>

</p:declare-step>
```

The first variant uses a connection between the primary output of the step named `load-xml` with the primary input of the XSLT step, while the second variant serializes the `p:load` steps in reverse order and therefore needs to insert a `p:sink` step in between, and it needs to connect the primary input of the XSLT step, `source`, explicitly with the primary output, `result`, of `load-xml`.

Unless one wants to use the x-y-coordinates as an ordering hint, the graph editor does not provide clues about the preferred serialization order for the `p:load` steps. One possibility that was quickly rejected was to use another type of connectors in the 2-D graph that represent document order. The idea was rejected because it puts an additional burden onto the user that seems unnecessary.

Generating a serialization that makes maximum use of primary port connections is an optimization problem. The author addressed it by a graph traversal that favours serializing primary ports adjacently when there are multiple choices. This is a heuristics that does not guarantee optimal results but solves this issue well enough.

The import mechanism has to deal with a problem that is also related to default readable ports: It needs to make implicit connections explicit for ports and also for options. The core of this problem has already been addressed in an XSLT-based XProc documentation tool whose normalized output will be converted to JointJS's internal JSON graph model using Saxon-JS.

Sub-Graphs

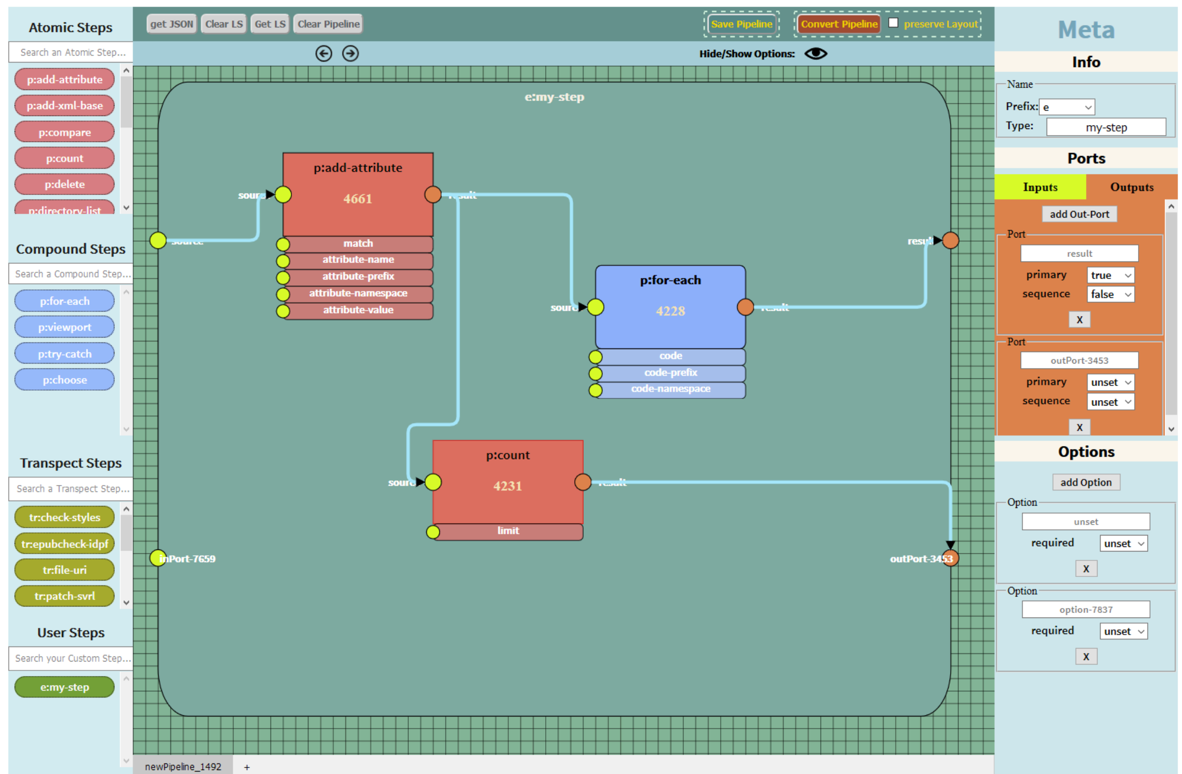
The subpipelines of compound steps such as `p:for-each` are displayed in their own tabs. Some bookkeeping in the Javascript application will make sure that they are included in the JSON representation upon export and that they are removed when the placeholder block in the containing graph is removed. JointJS would have permitted in-place folding of subpipelines, but expanding them would have quickly occupied much screen space and it would also have necessitated more advanced auto-layout capabilities.

Auto-layout is currently being added to the editor, its primary application being rendering the pipeline and its subpipelines (in separate tabs) initially after loading an existing pipeline.

For round-tripping (import, edit, export, import, ...), in order to spare users the ordeal of recreating a decent layout in xprocedit over and over again, there is an option to preserve layout information in the serialized pipeline, using XProc `p:pipeinfo` elements.

The application's user interface is still under development; in March 2019 it looked like Figure 2 [45].

Figure 2. User Interface



The step library palettes on the left are still supplied statically as JSON structures. There will be an import process that processes `p:import` statements and makes available the imported steps, grouped by namespace prefix. A difficulty that has already been solved is that imports often (in case of transpect, at least) use canonical import URIs that are not identical with their locations on the Web server that xprocedit runs on. A catalog resolver written in XSLT that runs in the browser will perform the required URI translations when recursively resolving the imports, and it will also restore the canonical URIs when serializing the XProc XML from the internal representation.

5. Other Visual XProc Editors

Even before XProc 1.0 was finalized, EMC published the interactive XProc Designer [EMCXProcDesigner] that ran in the browser. It was built using the Google Web Toolkit [GWT]. The editor was visually appealing, but lacked an important feature: It was not possible to import other steps or step libraries. This and other features are mentioned in a feature “pipeline” but development seems to have stalled since many years.

Another recently developed visual XProc editor is GProc [GProc]. It is written in Python with a GTK+ interface, therefore it does not run in the browser. Similar to xprocedit, it is the result of a master’s thesis.

6. Outlook

xprocedit has been written as part of one of the author’s master’s thesis.Much effort has gone into adapting the Javascript graph framework for XProc, therefore, given the limited amount of time available, some crucial features such as pipeline and library import are not functional yet. Refactoring some of the user interface components, such as the option editor, can use some rework. It is conceivable to use Saxon-JS to a larger extent for generating these types of forms.

While pipelines are currently “stored” in main memory, we will probably add a RESTXQ service, provided by a BaseX database, to store the edited pipelines and to load step libraries from.

If there is interest and funding, XProc can become a native graph type in JointJS or its commercial derivative, Rappid [JointJS].

If there is interest not only in editing pipelines in the browser but also in executing them in the browser, it is conceivable that, using Saxon-JS and interfacing other Javascript libraries, a subset of XProc 3.0 will be made available in the browser at some stage, probably as part of another master’s thesis.

An intermediate solution would be to run an XProc processor on a server and to post pipelines (and payloads) from the editor to the server via HTTP. A specific appeal of this solution is that both nascent XProc 3.0 processors, Calabash and Morgana, will accept alternative pipeline serialization formats than XML. So xprocedit might be able to post its internal graph representation as RDF or as its native JSON model, without the need to do the default readable port optimization that is only meant to simplify further editing in XML format.

7. Conclusion

A prototype of a visual XProc editor has been presented. Although the design choice for the graph library has been vindicated, a considerable amount of effort was necessary and will be necessary in order to add XProc as another supported graph type to the library, JointJS. Given that the visual editor will never been a complete programming environment (since much project-specific code will be written in XSLT, XQuery, schema languages, etc.) and XProc is a niche language, it is not clear whether xprocedit will be developed further or whether it will experience the fate of XProc Designer, whose development stalled shortly after the first release. At least the code is open source and can be picked up and modified by anyone.

Bibliography

[Blender] Blender Documentation: Node Editor https://docs.blender.org/manual/en/latest/editors/node_editor/

[BoldtSousa2012] Boldt Sousa, Tiago, (2012). Dataflow Programming: Concept, Languages and Applications https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf

[Boshernitsan1998] Boshernitsan, Marat, Downes, Michael (1998). Visual Programming Languages: A Survey https://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html

[EMCXProcDesigner] EMC XProc Designer https://community.emc.com/docs/DOC-4382

[GeueMT] (2019). Entwicklung eines grafischen Editors für XProc-Pipelines mit dem SVG-basierten JavaScript-Framework JointJS Master's Thesis, Hochschule Merseburg

[GProc] GProc – A Graphical Authoring Tool for XML Pipelines https://gitlab.com/in2erval/gproc/

[GWT] Google Web Toolkit (GWT) https://opensource.google.com/projects/gwt

[JointJS] JointJS https://www.jointjs.com/

[Quin2019] (2019). XProc in XSLT: Why and Why Not. In: XML Prague 2019 Conference Proceedings, p. 255. http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf

[Serialization31] XSLT and XQuery Serialization 3.1, W3C Recommendation, 21 March 2017 https://www.w3.org/TR/2010/REC-xproc-20100511/

[transpect] transpect, An Open-Source Framework for Converting and Checking Data https://transpect.github.io/

[XProc1] XProc: An XML Pipeline Language, W3C Recommendation, 11 May 2010 https://www.w3.org/TR/xslt-xquery-serialization-31/

[XProc3] XProc 3.0: A Pipeline Language http://spec.xproc.org/

[xprocedit] xprocedit https://github.com/mag-letex/xprocedit/

Generating documents from XQuery annotations

Andy Bunce, Quodatum Ltd

Abstract

The paper describes an implementation of an xqDoc.org documentation generator. A focus of this implementation is XQuery annotation support. In 2014, the xqDoc schema was updated to include markup to capture XQuery annotations; however, existing renderers have often not been updated to make use of this. A major driver for annotation support is documenting XQuery web applications built using the RESTXQ standard. RESTXQ defines a standard set of XQuery annotations that can be used to define RESTful Web Services from XQuery. Annotations are also being used to define frameworks for unit testing, user permissioning and web socket interfaces. Annotations are code markup that the runtime environment may choose to use to wire-in additional external functionality to XQuery applications. The xqDocA implementation is open source. It is largely written in XQuery and runs with recent versions of BaseX. It generates static, standalone HTML5 and XML and JSON output.

The included XQuery library modules can also be used to assist in the generation of other related documentation artefacts. In the case of RESTXQ, these could be the generation of openAPI (Swagger) and WADL documents.

1. Introduction

Making sense of an XQuery code base can be hard, even if you have written it yourself. It can be difficult to see how the parts fit together. This paper describes an attempt to build a documentation tool to help with that process with the particular goal of incorporating information from XQuery annotations. The paper looks at the history of annotations and the history of XQuery documentation tools.

2. Annotations?

What are they, and how are they used?

2.1. What are annotations?

Wikipedia defines an annotation as "a metadatum (e.g. a post, explanation, *markup*) attached to location or other data." Their use in mainstream software engineering began in 2004 when Java 5 introduced them via JSR 175.

Annotations were added as part of XQuery 3.0 in 2014. The syntax is a "%" followed by an EQName and optionally a set of values. The specification simply states: "XQuery uses annotations to declare properties associated with functions (inline or declared in the prolog) and variables. "

2.2. Use of annotations in XQuery

The following sections show some applications of XQuery annotations, which often mirror similar usage in Java frameworks.

2.2.1. Built-in annotations

The XQuery 3.1 standard defines only two annotations: `%private` and `%public`. XQuery 3.0 Update defines only one: `%updating`. It also states:

"Implementations may define further annotations, whose behaviour is implementation-defined. For instance, if the `eg` prefix is bound to a namespace associated with a particular implementation, it could define an annotation like `eg:sequential`. If the namespace URI of an annotation is not recognized by the implementation, then the annotation is ignored. Implementations may also provide a way for users to define their own annotations."

2.2.2. RestXQ

In 2012, Adam Retter presented a use of annotations to define a web interface [RESTXQ]. It is based on the Java standard RESTful Web Services [JAX-RS]. RESTXQ has since been implemented by many XQuery products as it provides a straightforward way to wire-up XQuery code to a web interface.

A simple RESTXQ example:

```
declare
%rest:path("hello/{$who}")
%rest:GET
function page:hello($who) {
  <response>
    <title>Hello { $who }!</title>
  </response>
};
```

Here the `%rest:path` annotation specifies a URL and the `%rest:GET` specifies an HTTP method. In a suitable Web server environment, a request for "/hello/fred" will be wired up to invoke the `page:hello` function with the argument "fred". The result of the function will be returned by web server.

2.2.3. Unit testing

Unit testing is another popular domain for annotations. In the Java world, JUnit has long made use of annotations. Many XQuery vendors provide a feature whereby tests can be run by invoking a command that scans a directory for modules containing functions marked with a custom annotation, such as `!unit:test`. An example of this is XRAY for MarkLogic [6].

2.2.4. Other applications

The BaseX product has recently extended its list of built-in annotation handlers with:

- `%ws` To define access to Web sockets [1]
- `%perm` To define a web application permission layer [2]

BaseX also defines annotations for lazy evaluation and locking [3]. The eXist-db product uses annotations in its HTML templating feature [4] and the MarkLogic product uses annotations for transaction control [5].

3. XQuery documentation

The following sections describe XQuery documentation formats and tools.

3.1. The xqDoc format

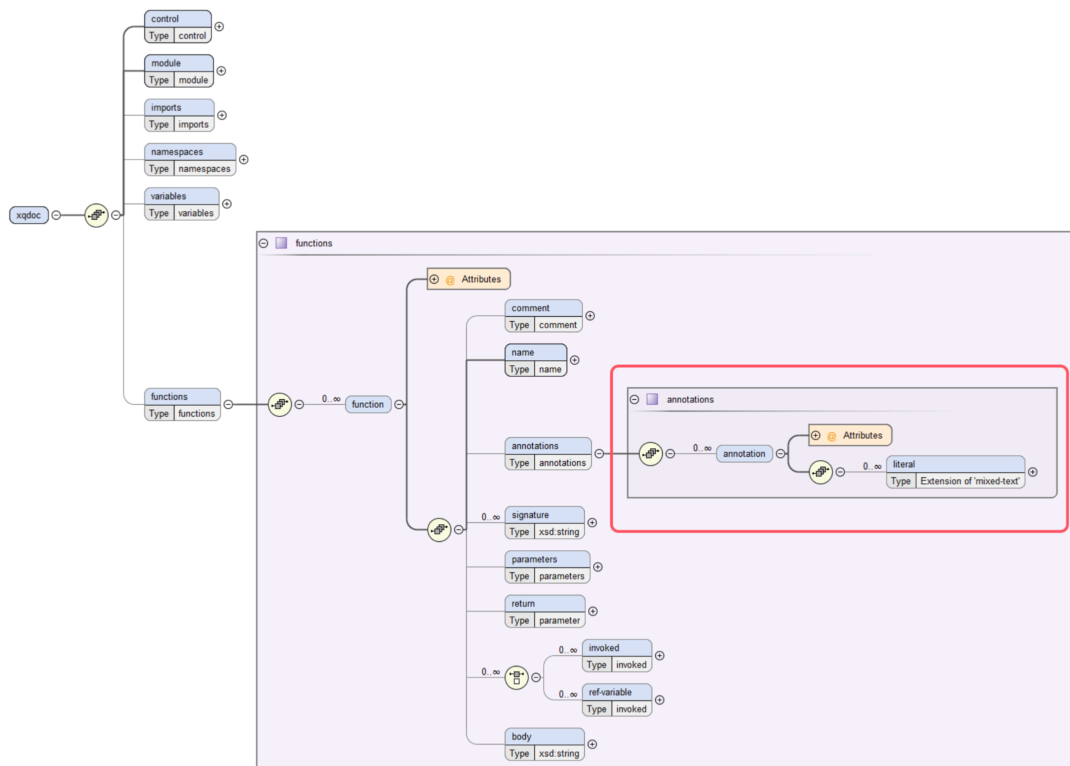
In 2002, Darin McBeath realised the need for a tool for generating documentation from XQuery sources and created the xqDoc website [XQDOC]. This site defines an XML schema for the namespace http://www.xqdoc.org/1.0. This schema provides a vocabulary to describe an XQuery module and a reference Java implementation.

To get full value from xqDoc, the source should contain comments formatted according to certain conventions in a similar fashion to Javadoc comments. See example below:

```
(:~
: The controller for constructing the xqDoc HTML information for
: the specified library module. The following information for
: each library module will be generated.
: <ul>
: <li> Module introductory information</li>
: <li> Global variables declared in this module</li>
: <li> Modules imported by this module</li>
: <li> Summary information for each function defined in the module</li>
: <li> Detailed information for each function defined in the module</li>
: </ul>
:
: @param $uri the URI for the library module
: @param $local indicates whether to build static HTML link for offline
: viewing or dynamic links for real-time viewing.
: @return XHTML.
:)
define
function print-module($uri as xs:string, $local as xs:boolean) as element()*
```

3.2. Schema updates

xqDoc was initially written to target XQuery 1.0. In 2014, the schema was extended to capture XQuery annotation information defined in XQuery 3.0. The figure below highlights the addition:



3.3. Working with xqDoc documents

The following features should be borne in mind:

Function names may be referenced in XQuery in a number of different styles. For example:

- `abc:foo(42)` or `prefix2:foo(42)`
- `foo(42) (: with a default function namespace definition :)`
- `Q{http://nowhere.com/funs}foo(42)`

Most xqDoc implementations return these as coded, rather than normalising them to a standard form. This can make finding cross-references more complex.

Some implementations allow their library modules to be used without explicit `import module` statements.

If these imports are not listed in the generated xqDoc, it can be difficult to resolve some names. This suggests an xqDoc generator needs information about the target platform both for its grammar and its static context.

3.4. Components

Implementations of xqDoc have three main components:

1. A component to parse the XQuery source code and generate the corresponding xqDoc XML elements for functions and variables, etc.
2. A component to parse xqDoc style comments `(:~ … :)` into the corresponding xqDoc.
3. Optionally, a means to render the resulting XML into formats for reading, such as HTML. This part typically uses XSLT.

3.5. XqDoc implementations

A partial list of xqDoc implementations is below.

Table 1. xqDoc implementations

Name	Ref	Language	Parser style	Last update	Notes
xqDoc	[XQDOC]	Java	ANTLR 2.7	2014	
xquery/ xquerydoc	[13]	XQuery	REX	2016	Also supplies an XProc step.
eXist-db	[15]	Java	(unknown)	Active	Supplied as library function.
BaseX	[16]	Java	(unknown)	Active	Supplied as library function.
wcanillion/ xqlint	[9]	Javascript (Node)	REX	2018	Use the help to find xqdoc option.
lcahlander/ xqdoc	[17]	Java	ANTLR 4	Active	

3.6. Parsers

Typically, the code to implement a parser is generated using a tool rather than hand written. Tools that have been used for the task of parsing XQuery are ANTLR [7] and REx [8].

REx has the valuable feature that its input format is EBNF. EBNF is the grammar language used in the XQuery standards. REx can also output parser code in many languages, including XQuery 1.0 and Javascript.

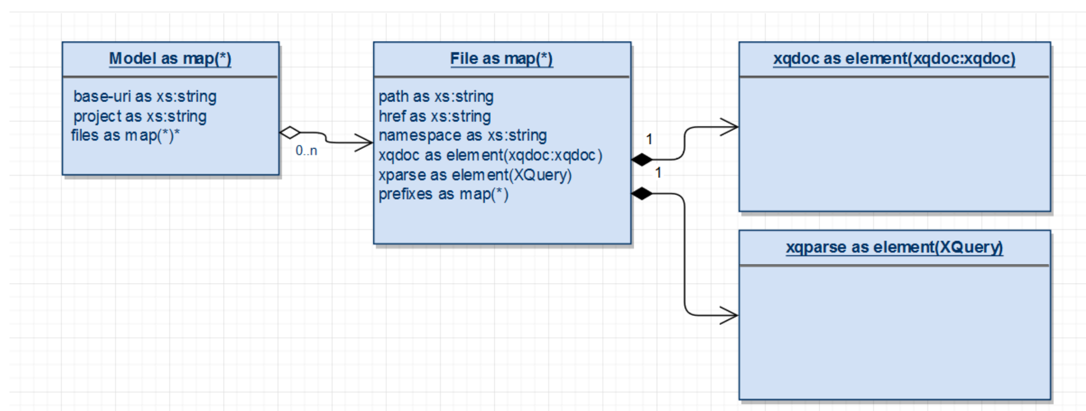
4. Introducing xqDocA

xqDocA was written to show the connections between various modules, as well as perspectives onto an XQuery code base.

4.1. Overview

Given a list of the files to process, xqDocA creates an XQuery map that holds information about the source location of each file along with a created XML parse tree and xqdoc output. This map is referred to as the *model*.

The model can be depicted as follows:



Another map, `$options`, holds configuration options to apply to the run. This map includes a list of "renderers" to run against the *model*.

A renderer creates a single output file. There are currently two kinds of renderers.

1. *global*: generate an output that is project wide, such as an index.
2. *module*: generate an output for each XQuery source file, such as a direct rendering of its xqDoc file.

4.1.1. Renderers

Renderers can be viewed as a map with the following keys:

Table 2. Renderer properties

Key	Type	Description
name	xs:string	Used to identify the renderer
description	xs:string	About the renderer
type	xs:string	"xqdoca:global" or "xqdoca:module"
uri	xs:string	The name of the output it creates
output	xs:string	Serialization parameters for the output
function	function(*)	The function that implements the rendering

The listing below shows some example renderers:

```
map {
  "output": "html5",
  "name": "index",
  "uri": "index.html",
  "function": Q{quodatum:build.xqdoc-html}index-html2#2,
  "type": Q{https://github.com/Quodatum/xqdoca}global,
  "description": "Index of sources"
}

map {
  "output": "json",
  "name": "swagger1",
  "uri": "swagger.json",
  "function": Q{quodatum:xqdoca.generator.swagger}swagger#2,
  "type": Q{https://github.com/Quodatum/xqdoca}global,
  "description": "Swagger file (JSON format) from restxq annotations."
}

map {
  "output": "xml",
  "name": "xqparse",
  "uri": "xqparse.xml",
  "function": Q{quodatum:xqdoca.mod-html}xqparse#3,
  "type": Q{https://github.com/Quodatum/xqdoca}module,
  "description": "xqparse file for the source module"
}
```

All outputs are created below the target folder specified in the `$options` map. For module style outputs, numbered sub-folders are created for each source module and the `uri` value is resolved relative to this folder.

Functions annotated as global renderers need to have arity=2 and signature `($model as map(*),$opts as map(*))`.

Functions annotated as module renderers need to have arity=3 and signature `($file as map(*),$opts as map(*),$model as map(*))`.

4.2. Implementation

The code runs under BaseX 9.2, and currently no XSLT is used by the built-in renderers.

For parsing XQuery and xqDoc comments, the ex-xparse component [18] is used. This component is a REX based parser. ex-xparse is modelled on John Lumley's proposal for an XParse Module [XPARSE]. It can parse a number of XQuery versions and dialects.

The rendering code is:

```
for $render in $global
let $doc:= apply($render?function,[$model,$opts])
return map{"document": $doc,
           "uri": $render?uri,
```

```
                "output":$xqo:outputs?($render?output)
              },

         for $render in $module, $file in $model?files
         (: override opts for destination path :)
         let $opts:=map:merge((
               map{
                 "root": "../../",
                 "resources": "../../resources/"
               }, $opts))
         let $doc:= apply($render?function,[$file,$opts,$model])
         return map{"document": $doc,
                    "uri": concat($file?href,"/",$render?uri),
                    "output": $xqo:outputs?($render?output)
                   }
               )
```

4.3. Sample outputs

MAIN / **MODULE**

quodatum:xqdoca.page

1	**Summary**
2	**Imports**
3	**Variables**
4	**Functions**
4.1	page:badge
4.2	page:calls
4.3	page:date
4.4	page:link-module
4.5	page:section
4.6	page:toc3
4.7	page:tree-list
4.8	page:view-list
4.9	page:wrap
5	**Namespaces**
6	**Restxq**
7	**Source**

4.3 page:date #

Arities: page:date#0 page:date#1

Summary
formated datetime

Signature

```
page:date ( ) as element(span)
```

```
page:date ( $when as xs:dateTime ) as element(span)
```

Return

- *element(span)*

▼ **Source**

```
function page:date()
as element(span)
{
 page:date(current-dateTime())
}
```

```
function page:date($when as xs:dateTime)
as element(span)
{
  <span title="{ $when }" >{ format-dateTime($when, "[h].[m01][Pn] on [FNn], [MNn] [D1o] [Y0000]")
}
```

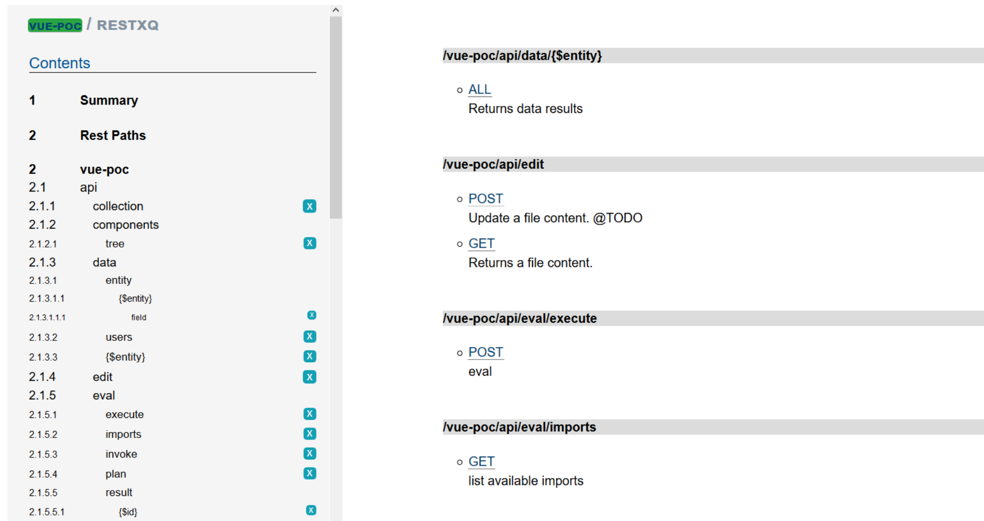
▸ **References 3 functions from 2 modules**

VUE-POC

1	**Summary**
2	**Modules**
2.1	Main modules
2.2	Library modules
3	**Files**
4	**Annotations**
5	**Other perspectives**

2.2 Library modules

Type	Uri		Annotations	calls
library	http://quodatum.com/ns/pipeline	U	http://www.w3.org/2012/xquery	29
library	http://quodatum.com/ns/pipeline			80
library	http://quodatum.com/ns/plantuml			28
library	quodatum.api.data	R	http://exquery.org/ns/restxq http://www.w3.org/2010/xslt-xquery-serialization	0
library	quodatum.data.history	U	http://www.w3.org/2012/xquery	3
library	quodatum.data.mimetype		http://basex.org	17
library	quodatum.data.tree			16
library	quodatum.dbtools	U	http://www.w3.org/2012/xquery	20
library	quodatum.model.rest	R	http://exquery.org/ns/restxq http://www.w3.org/2010/xslt-xquery-serialization	8



4.4. Customisation

With any report generator, it is likely that customisation will be required. To facilitate this, new xqDocA outputs can be added to the system without modifications to the driver code. A plug-in system is provided using the dynamic loading of code modules.

This is done using custom annotations within the xqDocA code base. Functions that generate xqDocA output must have annotations in the xqDocA namespace indicating their role. At runtime, a designated directory is scanned for XQuery modules, and functions containing the appropriate annotations can be invoked without requiring any other code changes.

Three annotations are defined in the xqDocA namespace, `"https://github.com/Quodatum/xqdoca"`.

Table 3. xqDocA annotations

Name	Description	Parameters
`xqdoca:global`	Indicates a global renderer	(name for reference, description)
`xqdoca:module`	Indicates a module renderer	(name for reference, description)
`xqdoca:output`	Serialization details	(output uri, serialization type)

Note

A function will have one of `xqdoca:global` or `xqdoca:module`, as well as an `xqdoca:output` annotation.

Sample usage:

```
declare
%xqdoca:module("module","Html5 report on the XQuery source")
%xqdoca:output("index.html","html5")
function xqh:xqdoc-html2($file as map(*),
                         $opts as map(*),
                         $model as map(*)
                        )
as document-node()
```

or

```
declare
%Q{https://github.com/Quodatum/xqdoca}global("swagger1",
                "Swagger file (JSON format) from restxq annotations.")
%Q{https://github.com/Quodatum/xqdoca}output("swagger.json","json")
function _:swagger($model as map(*),
                   $opts as map(*)
                  )
```

In order to enable runtime selection of code based on function annotations the following features must be provided by the XQuery environment:

1. Dynamic module loading. This is standardised in the XQuery 3.1 specification as `fn:load-xquery-module`. Currently, this feature is not widely supported; however, many implementation have custom variants that provide equivalent functionality.
2. Annotation introspection. That is the ability at runtime to determine what annotations are attached to a function. This is not a feature with any standardisation but again is widely supported via vendor libraries. See Saxon [10], MarkLogic [11], BaseX [12], and eXist-db [14].

5. Conclusions

The combination of maps with the "?" lookup operator and XML with XPath provides a powerful programming environment. The use of annotation driven plug-in modules provides a clean separation of concerns between the driving code and task at hand.

I hope that this paper may be of value for those working with XQuery code and perhaps inspire some new uses for annotations.

For code and issues, see https://github.com/Quodatum/xqdoca.

Bibliography

[XQUERY31SPEC] Josh Speigel: XQuery 3.1: An XML Query Language. 21 March 2017, W3C https://www.w3.org/TR/2017/REC-xquery-31-20170321/

[RESTXQ] Adam Retter: RESTXQ 1.0: RESTful Annotations for XQuery. 21 March 2016, http://exquery.org/ http://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html

[JAX-RS] Pavel Bucek: JSR 370: JavaTM API for RESTful Web Services (JAX-RS 2.1) Specification. 22 Aug 2017, Java Community Process (JCP) https://jcp.org/en/jsr/detail?id=370

[XQDOC] Darrin McBeath: xqDoc website. 13 Jan 2014, http://xqdoc.org http://xqdoc.org/index.html

[XPARSE] John Lumley: XParse Module. 8 Dec 2014, http://expath.org/ https://lists.w3.org/Archives/Public/public-expath/2015Feb/att-0003/xparse.html

[1] Christian Grün: Web sockets. BaseX http://docs.basex.org/wiki/WebSockets#Annotations

[2] Christian Grün: Web permissions. BaseX http://docs.basex.org/wiki/Permissions#Annotations

[3] Christian Grün: XQuery Extensions: Annotations. BaseX http://docs.basex.org/wiki/XQuery_Extensions#Annotations

[4] Wolfgang Meier: Templating. eXist-db https://exist-db.org/exist/apps/demo/examples/templating/templates.html

[5] Transaction annotations. MarkLogic https://docs.marklogic.com/guide/xquery/enhanced#id_94002

[6] Rob Whitby: An XQuery test framework for MarkLogic . https://github.com/robwhitby/xray

[7] Terence Parr: ANTLR. https://www.antlr.org/

[8] Gunther Rademacher: REx Parser Generator. https://www.bottlecaps.de/rex/

[9] William Candillon: JSONiq & XQuery Quality Checker . JSONiq & XQuery Quality Checker

[10] Michael Kay: Saxon function-annotations. https://www.saxonica.com/html/documentation/functions/saxon/function-annotations.html

[11] MarkLogic annotations. https://docs.marklogic.com/sc:annotations

[12] Christian Grün: BaseX inspect:function-annotations. http://docs.basex.org/wiki/Inspection_Module#inspect:function-annotations

[13] James Fuller, John Snelson: xquerydoc. https://github.com/xquery/xquerydoc

[14] Wolfgang Meier: xquery inspection. eXist-db https://exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/inspection

[15] Wolfgang Meier: docs:generate-xqdoc. eXist-db http://exist-db.org/exist/apps/fundocs/view.html?uri=http://exist-db.org/xquery/docs&location=/db/apps/fundocs/modules/scan.xql&details=true

[16] Christian Grün: inspect:xqdoc. BaseX http://docs.basex.org/wiki/Inspection_Module#inspect:xqdoc

[17] Loren Cahlander: lcahlander/xqdoc . https://github.com/lcahlander/xqdoc

[18] Andy Bunce: ex-xparse. https://github.com/expkg-zone58/ex-xparse

XQuery for Data Workers

Alain Couthures

Abstract

With some extensions, XQuery can be used to program Data Workers to manipulate data in various formats and in different environments.

1. Introduction

It always starts with a simple task that one simple script will surely satisfy easily. Depending on multiple environment constraints, there are, then, more and more scripts, written in different and, possibly, multiple programming languages, doing similar actions with more and more instructions in them.

Procedural programming languages are quite verbose: step by step, loops after loops, handlers and variables are necessarily created and modified. This is still more or less how processors effectively act.

Hopefully, there are much more concise programming languages where developers just describe what they want to get. XQuery is one of them. Benefits are readability and maintainability, even for non-programming consultants or domain experts.

It is surely interesting to evaluate XQuery as a unique script language and to identify required extensions for manipulating data. The best way to do that is to implement an XQuery engine for that purpose.

2. Requirements

A data worker is defined as a running program which manipulates data. It can get data from various sources then, possibly, transform it and save, or send, the resulting data. A common usage for data workers is building interfaces between applications, as ETL systems do at a larger scale.

Specifically, a data worker is to be a minimal program for a basic operation while multiple data workers can be active at the same time.

When data volume is not huge and processing time can be deferred, for example not during active hours, performance is just to be considered when execution is perceived to be far too slow. Input/output operations are probably, anyway, the most time consuming ones.

Data workers should be deployable on different machines with different operating systems, even small ones such as RaspBerry Pis.

3. Chosen environment

3.1. NodeJS

NodeJS is an application which can be compared to a Java Virtual Machine but for Javascript instead. It is available for most operating systems on a wide range of platforms and it is a stable product with more and more features, versions after versions. It runs with just one single thread, like in browsers, but, with heavy use of events and callback functions, multitasking becomes possible.

There are various APIs to access resources such as "File System", "Net", "HTTP",... so it can be used to build complex applications without user interface, except in command line.

NodeJS also comes with its own native basic HTTP server: no complex configuration for security or performance but an ideal approach for small, dedicated, even not necessarily local, web servers listening to one different port each.

NodeJS can easily be extended using a package manager such as "npm". Unfortunately, using external packages creates multiple dependencies with potential support issues. Using "famous" packages can result as a lazy way to program basic operations with a lot of extra useless features and can slow execution. A good portable solution has to minimize external packages use.

3.2. Browsers

Browsers are a convenient solution to render data produced by Data Workers. While data to be rendered could be too big to fit in a single HTML page, XForms is the easiest framework to access it with filters.

XForms can also be efficiently used to enter rich parameters for a Data Worker when CLI is not enough or when users need access from different machines.

3.3. XQuery

XQuery 3.1 can extract and transform data from XML documents but also from any text files, including JSON ones. XQuery Update Facility adds expressions for modifying data without having to build new data.

Native XML database implementations come with many function libraries to extend XQuery allowing to interact with operating systems and to communicate with others.

XQuery can be defined as an extension of XPath allowing to construct nodes and perform more complex operations. As a consequence, XQuery can construct data as any Server Page solution (ASP, JSP): XML documents, XHTML+SVG +XForms pages,...

4. Fleur: an XQuery implementation in Javascript

Fleur is an XQuery 3.1+XQuery Update Facility 3.0 implementation. It is written in vanilla Javascript allowing it to run, both, in browsers and with NodeJS.

Fleur includes its own DOM3 engine. It can also use browsers DOM for manipulating HTML pages, when used in "XQuery-in-the-browser" mode.

Fleur, primarily, was intended just to replace XSLTForms XPath 1.0 engine for XForms 2.0 support. It will do much more for XForms: no need for XForms 2.0 specific functions to construct nodes and XQuery Update Facility as a replacement for XForms actions XML notation.

Using the native HTTP server provided by NodeJS, Fleur can be used to fully develop small XForms-REST-XQuery (XRX) applications.

Fleur compiles XQuery expressions into a Javascript array of arrays. This Javascript array is an exact representation of the corresponding XQueryX notation. Compiling expressions is interesting when an expression is to be evaluated repeatedly (which is the case, for example, with XForms).

Fleur always evaluates expressions asynchronously (Javascript Promises) because it is a requirement for calls to functions such as doc() which can be located deeply within the XQueryX tree. It is also necessary not to use the single execution thread for too long time: periodically, Fleur allows other treatments to get access to the thread. At client side, a browser will not freeze and concurrent evaluations will terminate independently. At server side, the native HTTP server, for example, will start to treat another request when the current one is too time consuming.

It is possible to call any Javascript function in expressions prefixing with "js:".

Internally, Fleur is, by design, always manipulating nodes. Extra node types are defined: MAP and ENTRY, ARRAY, SEQUENCE, FUNCTION. Atomic values are stored in TEXT nodes with schemaTypeInfo associated.

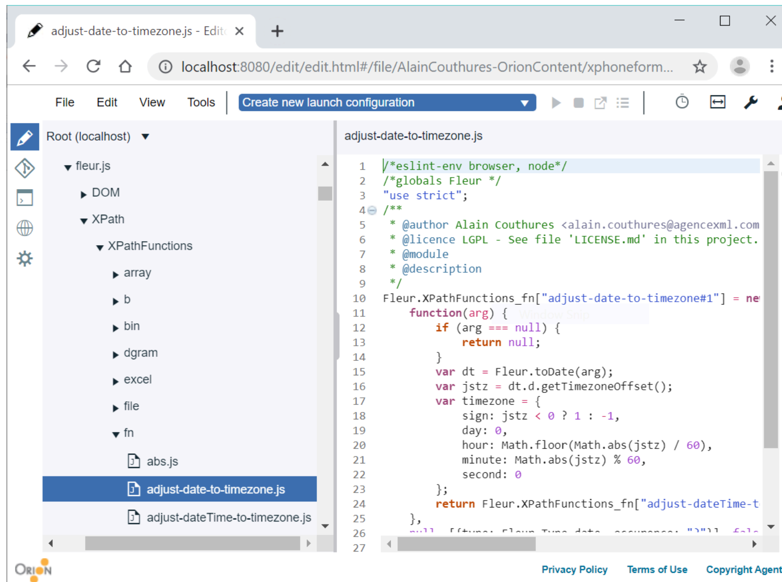
Fleur can be launched from command line with usual parameters using the node program.

```
C:\Users\Alain Couthures>node fleur --help
Usage: node fleur ([-s:xmlfile] [-o:outfile] (-q:queryfile|-qs:querystring)
                   [params]|[-p:port] [-f:folder])
 -s:     XML input file (optional)
 -o:     output file (optional)
 -q:     query file
 -qs:    query string
 -p:     http server port
 -f:     http server folder
 params  name=value as externals
```

5. Development and tests

As XSLTForms, Fleur is developed using Eclipse Orion which provides automatic sources checking while typing. It prevents from syntax errors, undeclared variables, unreached statements, unbalanced parentheses or curly braces. It is an important feature because Javascript is an interpreted programming language.

Figure 1. Eclipse Orion



There is a source file for each feature. The HTTP server is used to collect and concat single files using virtual URLs.

XPath functions are written in vanilla Javascript instructions. This is allowed by a convert mechanism for atomic values and sequences as parameters or returned value.

```
Fleur.XPathFunctions_prof["sleep#1"] =
    new Fleur.Function("http://basex.org/modules/proc", "prof:sleep",
 function(ms, callback) {
  if (ms > 0) {
   setTimeout(function() {
    callback(null);
   }, ms);
   return;
  }
  callback(null);
 },
 null, [{type: Fleur.Type_integer}], false, true, {type: Fleur.EmptySequence});
```

Tests are mainly performed with a browser. Thanks to virtual URLs, the file fleur.js is automatically refreshed from last edited source files. XForms, with XSLTForms, is used to run the official XQuery Test Suite, category per category then to allow unit test for debugging. Even if XSLTForms is not yet using Fleur for itself, because of events support in XForms, it can evaluate XQuery expressions asynchronously: in the same category page, tests are evaluated in parallel and the corresponding results are provided depending on the evaluation time.

Figure 2. XQuery Test Suite

Serialization for tests is specific: resulting values are serialized as an equivalent XQuery expression instead of just as string values. It guarantees no ambiguity about sequences, types and allows to check node types.

Figure 3. Fleur Sandbox

6. Extensions

6.1. The generalized doc() and serialize() functions

The doc() function is to be used to get any document of any type from any source.

The default scheme is, at browser-side, "http" and, at server-side, "file". A scheme "cmd" is to be added to also get data from local process execution.

An optional second parameter, similar to serialization options, allows to specify non-XML media-type and format specific parameters such as field separator for CSV data. When not present, the file extension, if there is one, is used to get an implicit media-type.

An optional third parameter will be added to specify a node containing the grammar to be considered for Invisible XML processing.

Non-XML data is automatically parsed by the doc() function. The resulting document can be navigated with the corresponding node types mapping. There is not necessarily an XML tree for non-XML data. For example, a .xlsx file, when parsed, is proposed as a map with, as entries, all the files within the ZIP format while Markdown text is proposed as the corresponding XHTML node tree.

The serialize() function has a similar optional second parameter to, possibly, serialize from one notation to another.

6.2. Two-dimensional sequences for tabular data

Tabular data from CSV or spreadsheets is treated a sequence of raws, each line being also a sequence. This implied yet another node type to distinguish from one-dimensional sequences. The separator ';' is defined to separate raws within a two-dimensional sequence.

When data comes from a .xlsx file, the excel:values function can extract a two-dimensional sequence from a sheet specifying the desired range.

The !! operator has been added to get raws one after one as the ! operator is to be used to get items one after one.

Headers can be associated to columns in a two-dimensional sequence.

It is convenient to be able transpose a two-dimensional sequence with the matrix:transpose() function. For example, a list of items, one per line, can be loaded in CSV format to obtain a vertical sequence of items then transpose it into a one-dimensional sequence.

Arithmetics could be added in the future.

6.3. Function Modules

XPath 3.1 specifications do not include functions, for example, to manipulate files and folders or send an HTTP request.

Native XML database solutions provide many modules for various purposes. Because of the variety of them. BaseX has been considered as a reference for implementing function modules in Fleur.

6.4. Server-side evaluation

Fleur can be asked to listen to a port number. It will then act as a basic HTTP server which will also execute .xqy pages and send back the result. Of course, XForms pages for XSLTForms can be generated by Fleur when executing a .xqy request. It can be used to, for example, provide inline minimal XForms instances which can be interesting for better performance.

This server can process simultaneous requests because of the asynchronous evaluations.

6.5. Client-side evaluation

Because it is contained in just one .js file, Fleur can easily be used at client-side. Even if NodeJS is supporting some of latest ECMAScript syntax and new features, Fleur source is written for recent browsers.

Of course, client-side evaluation is also limited by security: accessing folders and files is restricted, cross-domain may not be allowed and it is not possible to run external processes.

Nevertheless, it is always a convenient way to try and debug more or less complex expressions, especially with result being serialized in XQuery notation. The browser debugger can help too but a good knowledge of Fleur sources and XQueryX is required to place breakpoints in instructions.

7. Examples of Data Workers with Fleur

7.1. Bank statements converted into CSV files

Some bank statements are transmitted using old OXF notation, which is a pre-XML notation while others are transmitted as .xlsx files.

To allow an accounting solution to import those statements, the Data Worker has to navigate an input folder and, for each bank file, convert its content. Each entry in a bank statement should become, because of double-entry bookkeeping system, two raws into the corresponding generated CSV file.

Because FLWOR expressions results can be seen as sequences of columns, each transaction is, first, transformed into 2 columns then, before serialization, the resulting matrix has to be transposed.

```
let $d := doc('ofxexample.ofx')
let $matr := matrix:transpose(
  for $s in $d//STMTTRN
  let $amount := $s/TRNAMT
  let $nmin := fn:abs(fn:min((xs:decimal($amount), 0)))
  let $nmax := fn:abs(fn:max((xs:decimal($amount), 0)))
  let $date := $s/DTPOSTED
  let $year := fn:substring($date, 1, 4)
  let $month := fn:substring($date, 5, 2)
  let $day := fn:substring($date, 7, 2)
  let $name := $s/NAME
  let $sdat := concat($day, '/', $month, '/', $year)
  return
    matrix:transpose(
      $sdate, '471000', $name, $nmin, $nmax, 'E';
      $sdate, '512100', $name, $nmax, $nmin, 'E')
    )
)
return file:write('ofxexample.csv',
  matrix:labels(('Date', 'Num cpte', 'Libelle', 'Debit', 'Credit', 'E'), $matr),
  map {'header': 'present', 'media-type': 'text/csv', 'separator': ';'})
```

7.2. IT Inventory dashboards

There are many computers in an IT inventory. Each one has a name and textual properties with limited possible values (OS version, disk type,...).

The Data Worker is an HTTP server which receives properties from each computer and generates dashboards for the IT manager. There is a dashboard for each property. A dashboard is rendered as an HTML table where computer names are grouped per values for the corresponding property. While, for example, migrating for one OS version to another, the OS dashboard explicitly lists which computers are still to be migrated.

```
declare namespace output="http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:indent "yes";
processing-instruction xml-stylesheet {'href="xsl/xsltforms.xsl" type="text/xsl"'},
let $doc := fn:doc('../private/inventory.xml')/inventory
let $m := $doc/computers/computer
let $nicm := map {
  for $n in $doc/nics/nic
  return entry {fn:data($n/@idref-to-nic-owner)} {fn:data($n/timestamp)}
}
let $m2 := (
  for $s in $m
  order by $s/name
  return $s
)
```

```
let $totalPC := fn:count($m)
return
  <html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:xf="http://www.w3.org/2002/xforms">
    <head>
      <title>R&eacute;partition Postes</title>
      <style>
        body {{
          font-family: arial,sans-serif;
          font-size: 80%;
        }}
        table {{
          border-collapse: collapse;
        }}
        table, th, td {{
          border: 1px solid black;
        }}
      </style>
      <xf:model>
        <xf:instance>
          <inventory xmlns="" total="{$totalPC}" selected="version">{
            ('version', 'architecture', 'model', 'disktype', 'office') !
              <group name="{.}">
              {
                let $n := .
                for $s in $m2
                group by $v := (if ($s/*[name() eq $n])
                               then $s/*[name() eq $n]
                               else ' ')
                order by $v
                return
                  <item name="{if ($v eq ' ') then 'unknown' else $v}"
                        total="{fn:count($s)}">{
                    $s ! name ! xs:string(.)
                  }</item>
              }
            </group>
          }</inventory>
        </xf:instance>
      </xf:model>
    </head>
    <body>
      <p>{
        fn:format-dateTime(file:last-modified('../private/inventory.xml'),
          '[h01]:[m01]:[s01] [D]/[M]/[Y,2-2]') || ' - ' || $totalPC ||
          ' ordinateurs dans l''inventaire'
      }</p>
      <xf:select1 ref="@selected">
        <xf:itemset ref="../group">
          <xf:label ref="@name"/>
          <xf:value ref="@name"/>
        </xf:itemset>
      </xf:select1>
      <table>
        <tbody>
          <xf:repeat ref="group[@name = /inventory/@selected]/item">
            <tr>
              <td><xf:output value="if(@name = 'unknown', 'inconnu', @name)"/></td>
              <td><xf:output value="@total"/></td>
              <td><xf:output value="."/></td>
            </tr>
          </xf:repeat>
```

```
      </tbody>
    </table>
  </body>
</html>
```

7.3. XLSForm to XSLTForms

When it comes to write plenty of similar simple forms, it might be easier for authors to list items to be edited in a spreadsheet. It is possible to write .xlsx files in XLSForm format to obtain XForms pages for ODK. ODK is not a fully compliant XForms implementation.

The Data Worker has to read content from a .xlsx then generate the corresponding XForms page for XSLTForms. This can be done dynamically at server-side.

```
declare function local:setattr($name, $value) {
  let $esc := replace(
    replace(
      replace(
        replace(
          replace($value, '&amp;', '&amp;amp;'),
        '&lt;', '&amp;lt;'),
      '&gt;', '&amp;gt;'),
    '&quot;', '&amp;quot;'),
   "&apos;", '&amp;apos;')
  return if ($value ne '') then
   (if (contains($esc, '&amp;quot;')) then
     (' ' + $name + "='" + $esc + "'")
    else (' ' + $name + '="' + $esc + '"')) else ''
};
declare function local:attrs() {
  local:setattr("name", ?name) +
  local:setattr("label", ?label) +
  local:setattr("hint", ?hint) +
  local:setattr("calculation", ?calculation) +
  local:setattr("appearance", if (?type eq 'begin_group' and ?appearance ne '')
     then ('collapsed ' + ?appearance) else ?appearance) +
  local:setattr("relevant", ?relevant) +
  local:setattr("constraint", ?constraint) +
  local:setattr("constraint_message", ?constraint_message) +
  local:setattr("readonly", if (?type eq 'note') then 'true' else ?readonly) +
  local:setattr("required", ?required)
};
declare function local:attrs_choices() {
  local:setattr("list_name", ?list_name) +
  local:setattr("list_name", ?('list name')) +
  local:setattr("name", ?name) +
  local:setattr("label", ?label) +
  local:setattr("image", ?image)
};
declare function local:attrs_settings() {
  local:setattr("form_title", ?form_title) +
  local:setattr("form_title", ?title) +
  local:setattr("form_id", ?form_id) +
  local:setattr("default_language", ?default_language)
};
let $book := doc('public/grid.xlsx')
let $root := 'grid'
let $survey := excel:values($book, "survey!", (), true())
let $choices := excel:values($book, "choices!", (), true())
let $settings := excel:values($book, "settings!", (), true())
let $xlsform := '<xlsform>' +
  '<survey>' + string-join(matrix:transpose($survey !! (
  if (?type eq '') then '' else
```

```
  if (?type eq 'begin_group') then
    ('<group' + (if (?appearance eq '') then ' appearance="collapsed"' else '')
     + local:attrs() + '>') else
  if (?type eq 'end_group') then '</group>' else
  if (starts-with(?type, 'select_one ')) then
    ('<select_one choices="' + substring-after(?type, 'select_one ') +
     '"' + local:attrs() + '/>') else
  if (starts-with(?type, 'select_multiple ')) then
    ('<select_multiple choices="' + substring-after(?type, 'select_multiple ') +
     '"' + local:attrs() + '/>') else
    ('<' + ?type + local:attrs() + '/>')))) +
  '</survey>' +
  '<choices>' + string-join(matrix:transpose($choices !! (
  if (?('list name') eq '') then '' else
  ('<choice' + local:attrs_choices() + '/>')))) +
  '</choices>' +
  '<settings>' + string-join(matrix:transpose($settings !!
  ('<setting' + local:attrs_settings() + '/>'))) +
  '</settings>' +
  '</xlsform>'
let $doc := parse-xml($xlsform)
let $leaf := function($n) {
  element {$n/@name} {}
}
let $subtree := function($n, $t, $l) {
  element {$n/@name} {
    $n/* ! (if (name(current()) eq 'group')
            then $t(current(), $t, $l)
            else $l(current()))
  }
}
let $begin := '${'
let $end := '}'
let $refconv := function($n, $s, $b, $e, $f, $g, $r) {
  if (contains($s, $b)) then
   (substring-before($s, $b) + ' ' +
     $g($n, substring-before(substring-after($s, $b), $e), $r) + ' ' +
    $f($n, substring-after($s, $e), $b, $e, $f, $g, $r)) else
    $s
}
let $refpath := function($n, $name, $r) {
  let $target := $n/ancestor::survey//*[string(@name) eq $name]
  return '/' + string-join(($r,
              (reverse($target/ancestor-or-self::*[@name]) ! string(@name))), '/')
}
let $bind := function($n, $b, $e, $f, $g, $r) {
  if (name($n) eq 'group') then () else (
    let $type := (if (name($n) = ('text', 'note', 'select_one', 'select_multiple'))
                  then ()
                  else
      attribute type {'xsd:' + name($n)})
    let $xpattrs := $n ! (@required, @readonly, @relevant) !
                    attribute {name()} {if (string(.) eq 'true')
                                        then 'true()'
                                        else $f(., string(.), $b, $e, $f, $g, $r)}
    let $battrs := ($type, $xpattrs)
    return if ($battrs) then
     <xf:bind
       ref="{'/' + string-join(($r, (reverse($n/ancestor-or-self::*[@name]) !
                               string(@name))), '/')}">{$battrs}</xf:bind> else ()
  )
}
```

```
let $model := <xf:model>
  <xf:instance xmlns="">
    {element {$root}
      {($doc/xlsform/survey/* ! (if (name(current()) eq 'group') then
        $subtree(current(), $subtree, $leaf) else
        $leaf(current())),
      <meta>
        <instanceID/>
      </meta>)}
    }
  </xf:instance>
  {$doc/xlsform/survey//* !
   $bind(current(), $begin, $end, $refconv, $refpath, $root)}
</xf:model>
let $input := function($n, $r) {
  <xf:input ref="{'/' + string-join(($r, (reverse($n/ancestor-or-self::*[@name]) !
                                          string(@name))), '/')}">
    {$n/@appearance}
    {if ($n/@label ne '')
     then <xf:label mediatype="text/markdown">{$n/@label/text()}</xf:label>
     else ()}
    {if ($n/@hint ne '')
     then <xf:hint mediatype="text/markdown">{$n/@hint/text()}</xf:hint>
     else ()}
  </xf:input>
}
let $templates := map {
  'group': function($n, $m, $i, $r) {
      <xf:group ref="{'/' + string-join(($r,
                      (reverse($n/ancestor-or-self::*[@name]) !
                       string(@name))), '/')}">
        {$n/@appearance}
        {if ($n/@label ne '')
         then <xf:label mediatype="text/markdown">{$n/@label/text()}</xf:label>
         else ()}
        {$n/* ! (if ($m?(name(current())))
                then $m?(name(current()))(current(), $m, $i, $r)
                else $i(current(), $r))}
      </xf:group>
    },
  'note': function($n, $m, $i, $r) {
      <xf:output ref="{'/' + string-join(($r,
                        (reverse($n/ancestor-or-self::*[@name]) !
                         string(@name))), '/')}">
        {$n/@appearance}
        {if ($n/@label ne '')
         then <xf:label mediatype="text/markdown">{$n/@label/text()}</xf:label>
         else ()}
        {if ($n/@hint ne '')
         then <xf:hint>{$n/@hint/text()}</xf:hint>
         else ()}
      </xf:output>
    },
  'select_one': function($n, $m, $i, $r) {
      <xf:select1 ref="{'/' + string-join(($r,
                               (reverse($n/ancestor-or-self::*[@name]) !
                                string(@name))), '/')}">
        {$n/@appearance}
        {if ($n/@label ne '')
         then <xf:label mediatype="text/markdown">{$n/@label/text()}</xf:label>
         else ()}
        {if ($n/@hint ne '')
```

```
           then <xf:hint>{$n/@hint/text()}</xf:hint>
           else ()}
          {$n/ancestor::xlsform/choices/choice[string(@list_name) eq string($n/@choices)]
            <xf:item>
              <xf:label>{@label/text()}</xf:label>
              <xf:value>{@name/text()}</xf:value>
            </xf:item>
          }
        </xf:select1>
      },
    'select_multiple': function($n, $m, $i, $r) {
        <xf:select ref="{'/' + string-join(($r,
                          (reverse($n/ancestor-or-self::*[@name]) !
                           string(@name))), '/')}">
          {$n/@appearance}
          {if ($n/@label ne '')
           then <xf:label mediatype="text/markdown">{$n/@label/text()}</xf:label>
           else ()}
          {if ($n/@hint ne '')
           then <xf:hint mediatype="text/markdown">{$n/@hint/text()}</xf:hint>
           else ()}
          {$n/ancestor::xlsform/choices/choice[string(@list_name) eq
                         string($n/@choices)] !
            <xf:item>
              <xf:label>{@label/text()}</xf:label>
              <xf:value>{@name/text()}</xf:value>
            </xf:item>
          }
        </xf:select>
      }
  }
  let $view := $doc/xlsform/survey/* !
               (if ($templates?(name(current())))
                then $templates?(name(current()))(current(), $templates, $input, $root)
                else $input(current(), $root))
  let $form := document {(processing-instruction
                 xml-stylesheet {'href="xsl/xsltforms.xsl" type="text/xsl"'},
    <html
      xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xf="http://www.w3.org/2002/xforms"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <head>
         <title>{data($doc/xlsform/settings/setting/@form_title)}</title>
                {$model}
      </head>
      <body>{$view}</body></html>)}
  let $result := parse-xml(serialize($form, map{'indent': 'yes'}))
  return $result
```

7.4. Collecting from network equipments

Network equipments such as switches know which equipments are connected to them using MAC address tables associated to ports. This can be used to locate those equipments and it enables also to check that the switch configuration is the one required (VLAN, PoE, speed,...). Unfortunately, such tables have to be automatically purged in the network equipments. Luckily, there are network equipments with firmwares allowing to interact with them with a REST API using JSON.

The Data Worker has to periodically collect MAC tables using the REST API then transform and send the resulting data to another Data Worker acting as an HTTP server. The second Data Worker can then propose a dedicated dashboard.

```
for $i in 1 to 10000
let $step := (
  let $pcs := doc('collect.json')?*
```

```
  let $switches := doc('switches.json')?*
  let $login := doc('login.json')
  let $result := map {
    for $switch in $switches
    let $m := fn:doc('http://' || xs:string($switch?ip) ||
                     ':80/rest/v3/login-sessions',
                map {'method': 'json', 'http-verb': 'POST',
                     'timeout': '3000'},
                map {'userName': xs:string($login?user),
                     'password': xs:string($login?password)})
    return (if ($m?cookie) then (
      let $sessionId := map {'method': 'json', 'http-verb': 'GET',
                             'http-cookie': xs:string($m?cookie)}
      let $vps := doc('http://' || xs:string($switch?ip) ||
                      ':80/rest/v3/vlans-ports', $sessionId)?vlan_port_element?*
      let $macs := doc('http://' || xs:string($switch?ip) ||
                       ':80/rest/v3/mac-table', $sessionId)?mac_table_entry_element?*
      let $trk := $vps[?port_mode eq 'POM_TAGGED_STATIC' and ?vlan_id ne 5] !
                       xs:string(?port_id)
      for $port in $vps[not(?port_id = $trk)] ! ?port_id
      for $vlanid in $vps[?port_id eq $port] ! ?vlan_id
      let $portmacs := $macs[?port_id eq $port] ! ietf:mac(?mac_address)
      return if (not(empty($portmacs))) then (
        for $portmac in $portmacs
        let $pcname := local-name(head($pcs[ietf:mac(?mac) eq $portmac]))
        let $ename := (if ($pcname) then $pcname else '#' || xs:string($portmac))
        return entry {$ename} {map{'switch': local-name($switch),
                     'port': xs:string($port), 'vlan': xs:string($vlanid)} }
      ) else ()) else
      trace((), local-name($switch) || ': Connection refused. '))
  }
  let $t := (if (exists($result?*)) then (
    trace((),
      xs:string(current-dateTime()) || '    ' ||
                count(http:send-request(<http:request method='post'/>,
      'http://switchmanager:5000/batchcollect.xqy', $result)[2]/node()?*) ||
      ' MAC entries found. ')
    ) else
      trace((), xs:string(current-dateTime()) ||
           '    ' || 'No entry. '))
  let $pause := prof:sleep(1000 * 60 * 3)
  return ()
)
return ()

declare %updating function local:mergejsonfile($path, $batch) {
  let $doc := doc($path)
  let $m := $doc/map()
  let $update := (
    for $item in $batch?*
    return (if ($m?(local-name($item))) then
        replace node $m?(local-name($item))/node() with
          map:merge(($item/node(), $m?(local-name($item))/node()))
      else
        insert node entry {local-name($item)} {$item/node()} into $m))
  let $write := file:write($path, $doc, map {"indent" : "yes"})
  return $batch
};
let $filename := 'collect.json'
let $batch := request:body-doc()
let $tstamp := xs:string(current-dateTime())
let $addstamp :=
  (for $item in $batch?*
```

```
  return (insert node entry btimestamp {$tstamp} into $item/node(), ()))
return local:mergejsonfile($filename, $batch)
```

7.5. XForms 2.0 Test Suite for XSLTForms

The XForms 2.0 Test Suite can be downloaded as a .zip file. Each test already contains a processing instruction for the XSLTForms XSLT stylesheet to transform it but its path does not correspond to the one for the development environment of XSLTForms, which is the same as Fleur, with NodeJS.

The Data Worker has to unzip the Test Suite file, change the processing instruction in each test, change the file extension for each test file from .xhtml to .xml and update the index page accordingly.

7.6. Updating users accounts from HR software

Various applications need an up to date list of users. This can, usually, be obtained from the HR software. Depending on how an application allows or not to partially update its own list of users and how it stores historical data related to them, it can be required to minimize updates as just, for example, modified properties.

The Data Worker has to extract data from both systems, to compare them, to identify updates to be done and to format them accordingly to the targeted application. For example, it will generate Powershell commands to update an ActiveDirectory domain or generate REST API requests to delete and add members to Gmail groups.

8. Conclusion

This evaluation shows that it is very interesting to use XQuery for writing programs for Data Workers. The corresponding programs are short ones because all the mechanics to effectively access or post data are necessarily embedded at lower level: an XQuery implementation with such features allows programmers to concentrate to data transformation and XQuery is clearly powerful at this.

Fleur is not yet a mature implementation and, effectively, not yet fully compliant with XQuery specifications but it is now already used in production because of all those extensions for Data Workers. Performance are not always very good but there are already known optimizations to be added.

For Data Workers, there are more extensions now identified as interesting to be added: generate PDF files (possibly using XSL-FO notation), send and receive emails, run Tesseract OCR to get ALTO XML documents. The use of handlers has also to be improved.

Bibliography

[Fleur] Alain Couthures, *Fleur*, https://github.com/AlainCouthures/xphoneforms/blob/master/build/js/fleur.js

[XQuery] Robie, Jonathan, Michael Dyck, (eds.), *XQuery 3.1: An XML Query Language*, W3C, 2017, http://www.w3.org/TR/xquery-31/

[XPath functions] Michael Kay, (ed.), *XPath and XQuery functions and operators 3.1*, W3C, 2017, https://www.w3.org/TR/xpath-functions-31/

[xquf] Snelson, John and Jim Melton, (eds.), *XQuery Update Facility 3.0*, W3C, 2015, http://www.w3.org/TR/xquery-update-30/

[XF11] John M. Boyer, (ed.), *XForms 1.1*, W3C, 2009, https://www.w3.org/TR/2009/REC-xforms-20091020/

[XF2] E. Bruchez, *et al.*, (eds.), *XForms 2.0*, W3C, 2018, https://www.w3.org/community/xformsusers/wiki/XForms_2.0

[Pemberton 2013] Pemberton, Steven. "Invisible XML." Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In *Proceedings of Balisage: The Markup Conference 2013*. Balisage Series on Markup Technologies, vol. 10 (2013). doi:10.4242/BalisageVol10.Pemberton01.

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

[OFX] Open Financial Exchange. Downloads. http://www.ofx.net/downloads.html

[ODK] Open Data Kit. XLSForm Online. https://opendatakit.org/xlsform/

*sub*check Article MarkupUK London

Andreas Tai, Institut für Rundfunktechnik (IRT)

Michael Seiferle, BaseX GmbH

Abstract

The exchange of data in XML goes hand in hand with its validation. Mature technologies for this task exist. But often more is needed than just the technical application of XML Schema, Schematron or Relax NG. A user-friendly solution requires an abstraction from the XML foundation. Initiation and result of a validating process needs to be put into the context of the end users domain and be a guidance to solve conformance problems.

This paper shows how Schematron, XSLT, XQuery and JavaScript have been glued together to address these requirements. Although the subcheck framework (http://subcheck.io [http://subcheck.io/]) was implemented specifically for the Timed Text Markup Language (TTML) the approach can be used by any other XML vocabulary.

1. Intro

Do you share this problem? You have a well-documented XML structure and an XML Schema to test conformance, but:

- People do not use the XML Schema because they do not know it exists.
- People know the XML Schema exists, but they do not know how to apply it.
- People apply the XML Schema, but they do not know how to interpret the error messages.

If so: you are in the same situation as we were in 2012.

2. TTML and its Profiles

The W3C Timed Text Markup Language (TTML) [TTML2] is an XML vocabulary for video subtitles and captions.

Like many other XML vocabularies, TTML works as a baseline and covers many known use cases. One use case is the exchange of subtitle files, another use case is the rendering of subtitles in a video player. That's why it comes to no surprise that in operation not the complete TTML vocabulary is used, but only subsets of the Markup Language.

Examples of these subsets are EBU-TT-D [EBUTTD] defined by the European Broadcast Union (EBU) and the widely adopted TTML Profiles for Internet Media Subtitles and Captions (IMSC) [IMSC11] defined by the Timed Text Working Group (TTWG) [1]of the World Wide Web Consortium (W3C), that also published TTML.

3. Validation of TTML Profiles

Especially in the introduction phase of a new standard, it is important to make sure that the APIs of different software systems use the standards in the same way and conform to the rules of the specifications.

Once a workflow is established, the exchanged documents need to conform to the established APIs.

Take for example the following real-world use case scenario: a freelancer creates subtitles for a media service company which in turn got a contract from a streaming company to deliver subtitles for a complete season of a video series.

The media service company and the streaming company agree on IMSC as the technical baseline for subtitle XML files, but they may also apply their own set of "house style rules". So at the gateways of the service company and the streaming provider, overlapping but also separate checks are executed.

The Institute für Rundfunktechnik (IRT) [2]was well aware of this and other scenarios such as the "in-house" exchange of subtitle files and aimed for a better support for them.

In 2013, the IRT had already been active in TTML standardization, especially in the definition of the EBU-TT profiles.[3] But although W3C XML Schema (XSD) files existed, non-conformant documents were still produced and distributed.

This was due to a lack of knowledge about the existence of the XSD itself, its application or about the meaning of the implied rules and resulted in the new goal to build a more user-friendly validation alternative. The main inspiration came from the W3C HTML validation service.

In the process of the installation of their own service, the IRT went through different stages:

- A master thesis on the topic was written by Barbara Fichte (supervised by Prof. Dr. Anne Brüggemann-Klein) and a first prototype was implemented.
- A cooperation with the company BaseX GmbH was started.
- Full working solutions were implemented.
- Two different products (*sub*check and IMF Analyser) were launched to integrate the solutions.

In the following, we describe the implementation of the *sub*check service by BaseX GmbH and the IRT.

4. Validation Requirements

One important requirement was to validate a TTML XML document against different sets of conformance rules without duplicating the implementation effort, another that the end user should only get one error message when the same rule

[1]https://www.w3.org/AudioVideo/TT/
[2]The Institut für Rundfunktechnik GmbH (IRT) (Institute for Broadcasting Technology Ltd.) is the research centre of the German broadcasters, Austria's broadcaster and the Swiss public broadcaster.
[3]see https://tech.ebu.ch/groups/subtitling

of different specifications was broken. On top of that, the implementation of the software architecture was also guided by a number of non-functional requirements.

Team Expertise

The choice of technologies had to reflect the expertise available in the technical teams. For example, the IRT team worked intensively with XSLT, Schematron and W3C XML Schema but less with XQuery or Relax NG.

Deployment of Technologies

The technologies used needed to be widely deployed because it had to be possible to integrate the solution into different system environments.

All XML

We believed and still believe in the power of XML technologies. The cooperation of communities responsible for different XML technologies resulted in well-aligned XML technologies like XPath, XSLT, XQuery, Schematron and XPROC. We therefore wanted to use XML technologies wherever possible.

Separation of Concerns

We favoured a design approach with a framework of loosely coupled components that communicate over an agreed API. This way, different stakeholder groups can work on different parts of the framework without interfering with the work of other stakeholders.

We identified three different stakeholder groups:

- The domain stakeholders who know about conformance rules.
- The developers that implement the rules using validation technologies.
- The product team that technically designs and implements the end product.

End user Requirements

Existing validation strategies often come with very technical result messages, but users are often not technical. Even if they are, they may not be familiar with an XML schema technology or XML at all. Therefore, the validation should give information in an understandable way. The information should highlight not only what is wrong but also WHY something is wrong and how to correct it. It should enable the user to trace back the error to the documentation of a rule in the specification.

Product View

From the commercial side, some components needed to be integrated separately into different products. Different developer teams needed to work independently from each other.

5. Implementation Approach

5.1. Master Thesis

Barbara Fichte explored the topic in depth in her well written master thesis "Strategies for User-Oriented Conformance Testing of XML Documents".[MAFICHTE] She worked out the requirements, evaluated different schema languages and described how she implemented a first prototype.

She looked at the schema languages W3C XML Schema, Schematron, Relax NG and NVDL. The conclusion was that Schematron was the best fit for the purpose. One major aspect was the expertise in the team. But most importantly Schematron makes it easy to integrate additional information into the validation.

The other Schema technology that was favoured was W3C XML Schema. It is widely implemented and works well with grammar-based constraints. But W3C XML Schema came with a number of limitations. The most important one was that for some types of rules XML Schema 1.1 was needed and at the time of writing of the master thesis only limited support by free or open-source XSD 1.1 schema parsers was available.

As Schematron was such a good fit, Barbara Fichte proposed and implemented an approach where all constraints could be implemented with Schematron. She used Schematron's abstract patterns to implement grammar constraints that would

usually be implemented with W3C Schema. One example of these constraints is the order in which elements may appear under a parent element.

In the later implementation of *sub*check, the main schema technology used was indeed Schematron. However, we did not completely remove W3C XML Schema from the validation process. The costs to re-implement already existing grammar constraints in Schematron was too high compared to the benefit. There also existed some XML Schema of TTML profiles published by standard bodies. The use of these schemas enabled *sub*check validation to align better with the validation approaches of these organizations.

5.2. Application Implementation

In the final implementation, we separated three different steps:

- The documentation of the rules in a constraints XML document.
- The implementation of the rules in Schematron.
- The creation of a report that could be used by an end user product.

As the approach we took can be applied to any XML vocabulary in many scenarios, we will present a simplified use case to visualize the key points.

The scenario is to test cabin bag weight against the allowance of the two assumed airlines Aeto and Örn.

In the appendix, different examples show how the approach can be applied to TTML vocabularies.

5.2.1. Rules Documentation

The base of the *sub*check framework is the documentation of the rules in a separate XML file (constraints XML).

The implementation assumption is that more than one specification needs be tested. These specifications need to be defined first. To keep the XML samples in the paper short we later use only one specification.

```
<Specifications>
    <Specification ID="ID-Aeto-Conditions">
        <Name>Conditions Aeto</Name>
        <Acronym>C-Aeto</Acronym>
        <Version>1.0</Version>
    </Specification>
    <Specification ID="ID-Oern-Conditions">
        <Name>Conditions Örn</Name>
        <Acronym>C-Örn</Acronym>
        <Version>1.0</Version>
    </Specification>
</Specifications>
```

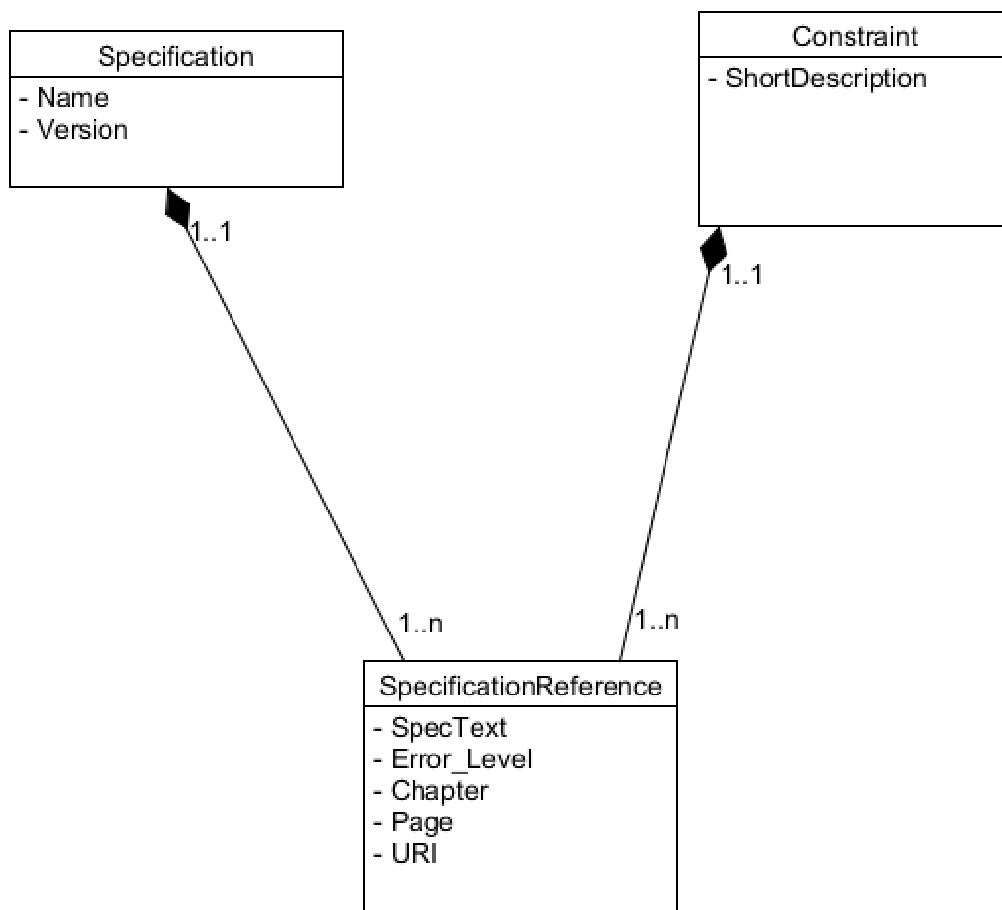
After that, the constraints can be documented with the necessary level of detail and linked back to the specification.

```
<Constraint ID="c1">
    <ShortName>
      Cabin Bag Max. Weight 8kg
    </ShortName>
    <SpecifiedBy ID="ID-cabinbag-8kg">
        <SpecificationReference>
          ID-Aeto-Conditions
         </SpecificationReference>
         <Error_Level>ERROR</Error_Level>
    </SpecifiedBy>
    <ShortDescription>
      Value should be equal or less than 8.
    </ShortDescription>
    <ShortDescriptionUser>
```

```
      Cabin bag should not have more than 8kg weight.
    </ShortDescriptionUser>
</Constraint>
```

The following UML class diagram models the relationship between constraints and specifications.

Figure 1. UML Diagram Constraint - Specification



All information documented in the constraints XML should be specific to the constraint and not its implementation.

One constraint can be part of different specifications and each of these specifications may put the constraint in a different context. While one specification may specify the constraint as a mandatory requirement, another specification may just regard it as recommendation. The first context would result in an error and the second context would result in a warning.

5.2.2. Schematron Schema File

The rule is implemented in Schematron as defined by the Schematron standard [ISOSCHEMA]. In addition, the `@see` attribute is used for linking the Schematron rule back to the documented constraint.

```
<sch:rule context="cabin-bag/weight">
    <sch:let
      name="bag-weight"
      value="xs:float(.)"/>
    <sch:let
      name="passenger-name"
      value="../../name"/>
```

```
    <sch:assert
      test="$bag-weight le 8"
      see="http://www.cabin-bag.info/c1"
      diagnostics="diag-weight-8">
        The weight of cabin luggage is 8kg
        or less.
    </sch:assert>
</sch:rule>
```

More context information is given by using Schematron's diagnostic element.

```
<sch:diagnostic id="diag-weight-8">
  The cabin luggage of
  <sch:value-of select="$passenger-name"/>
  exceeded the maximum weight allowance by
  <sch:value-of select="$bag-weight -8"/>kg.
  Pack lighter!
</sch:diagnostic>
```

5.2.3. Compiled Schematron

The Schematron Schema is compiled into an XSLT using a customized version of the Schematron skeleton implementation [4]. The sekeleton implementation needed to be adjusted for two reasons:

1. We also wanted to use attributes as rule context.[5]
2. We wanted to provide a more human readable version of the location where the error occurs.

Given the the following XML file...

```
<passenger>
    <name>Jane Grant</name>
    <cabin-bag>
        <weight>11</weight>
    </cabin-bag>
 </passenger>
```

...the target output of the compiled Schematron XSLT is an XML document that follows the structure of the Schematron Validation Report Language. A failed Schematron assert would result in the following SVRL:

```
<svrl:failed-assert
  test="$bag-weight le 8"
  id="assert-c1-1"
  see="http://www.cabin-bag.info/c1"
  location="/passenger[1]/cabin-bag[1]/weight[1]"
  subcheck:alternativeLocation="/passenger/cabin-bag/weight">
   <svrl:text>
     The weight of cabin luggage is 8kg or less.
    </svrl:text>
   <svrl:diagnostic-reference
      diagnostic="diag-weight-8">
       The cabin luggage of Jane Grant exceeded
       the maximum weight allowance by 3kg.
       Pack lighter!
   </svrl:diagnostic-reference>
</svrl:failed-assert>
```

The reference to the documentation of the constraint in the constraint.xml is kept in a `@see` attribute.

[4] see https://github.com/Schematron/schematron

[5] see also the discussion on issues https://github.com/Schematron/schematron/issues/44 and https://github.com/Schematron/schematron/issues/29 and pull request https://github.com/Schematron/schematron/pull/41

5.2.4. Generation of the Report

The SVRL is taken as input for the transformation into an XML report. This report is provided to the end user product that integrates the validation. The XML structure of the report is the agreed API between validation engine and post-processing system.

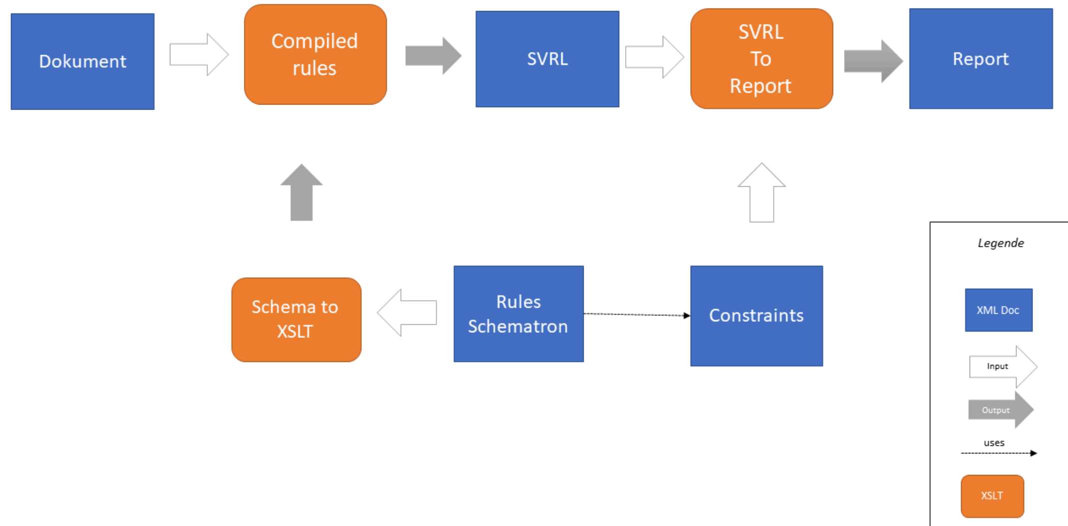
The output is grouped by constraint in the report.

```
<errorCategory>
    <constraintID>
      c1
    </constraintID>
    <title>
      Cabin Bag Max. Weight 8kg
    </title>
    <shortUserDesc>
      Cabin bag should not have more
      than 8kg weight.
    </shortUserDesc>
    <longUserDesc/>
    <specs>
        <spec>
            <name>
              Conditons Aeto, Version 1.0
             </name>
            <nameAcronym>
              C-Aeto<
            /nameAcronym>
            <errorLevel>ERROR</errorLevel>
        </spec>
    </specs>
    <errors>
        <error>
            <messages>
                <messageMain>
                  Assertion: The weight of cabin
                  luggage is 8kg or less.
                  Error Information: The cabin
                  luggage of Jane Grant exceeded
                  the maximum weight allowance
                  by 3kg. Pack lighter!
                </messageMain>
                <messageAssertion>
                  The weight of cabin luggage is
                  8kg or less.
                </messageAssertion>
                <messageDiagnosticsAll>
                  The cabin luggage of Jane Grant
                  exceeded the maximum weight
                  allowance by 3kg. Pack lighter!
                </messageDiagnosticsAll>
            </messages>
            <locations>
                <location
                  locationType="resolvableXPATH">
                  /passenger[1]/cabin-bag[1]/weight[1]
                </location>
                <location
                  locationType="humanXPATH">
                  /passenger/cabin-bag/weight
                </location>
            </locations>
```

```
        </error>
    </errors>
</errorCategory>
```

5.2.5. Overview: The Transformation Chain

Figure 2. Transformation in the Validation Chain



6. Interfaces: Machine-to-Machine and the End-User

*sub*check's core interfaces are based on RESTXQ and are implemented in the BaseX XML Server as a pure XQuery application.

The service contains a single endpoint `/validate` that orchestrates the validation engine and returns the resulting report. Files that are uploaded by users will not be persisted by the engine, they are parsed and validated in memory only and returned to the caller right away. The report will be returned in its XML form by default, but other formats are possible.

The XQuery implementation receives an XML file — it returns an error on non-XML files — and runs the XSL transformation for the compiled rules. The result of this process is then passed on to another stylesheet that transforms it to the reporting format and if necessary conducts conversions to other formats such as JSON or text.

In practice, a request looks like the following:

```
$ cat luggage.xml | http POST http://subecheck/validate
###################################################
> Content-Type: application/xml; charset=UTF-8


  <report>
    <errors>
    <constraintID>c1</constraintID>
    <title>
     Cabin Bag Max. Weight 8kg
    </title>
    <shortUserDesc>
      Cabin bag should not have more than 8kg weight.
    </shortUserDesc>
```

```
    ...
  </report>
```

Via REST, users can also request a report in JSON format by specifying a different `Accept`-Header:

```
$ cat luggage.xml | http POST http://subecheck/validate Accept:application/json
##########################################################################
> Content-Type: application/json; charset=UTF-8


{
  "filename": "luggage.xml",
  "filesize": "2 KB",
  "report": [{
    "constraintID" : "c1",
    "title"        : "Cabin Bag Max. Weight 8kg",
    "shortUserDesc": "Cabin bag should not have more than 8kg weight.",
    "longUserDesc" : "Assertion: The weight of cabin […]",
    "specs"        : [{
      "errorLevel" : "ERROR",
      "name"       : "Conditons Aeto, Version 1.0",
      "nameAcronym": "C-Aeto",
      "section"    : "[…]",
      "text"       : "[…]",
      "uri"        : "https://c-aeto/spec"
    }],
```

While *sub*check was designed with arbitrary frontends in mind — i.e. it can be easily integrated into existing workflows thanks to its almost universally accessible REST interface — we decided to implement a visual interface as a proof-of-concept that allows users to interactively explore and assess their validation results.

This interface heavily builds on the `JSON`-serialization of the reports and is implemented as a Single-Page-Application made up of `Vue.js`-components, that allow the user to potentially browse and filter hundreds of validation messages.

6.1. Using the *sub*check Application

In order to validate an XML file, users may initiate validation by dragging their file onto the browser window. Once the file is dropped, it is sent to the server and the validation pipeline will start. Once the pipeline has finished, usually in under a second, the result list will appear.

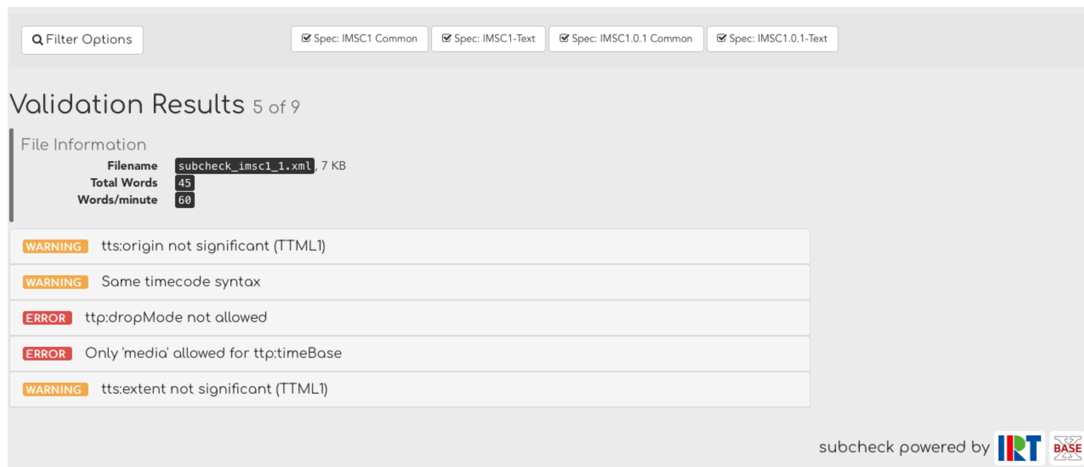
The following screenshots are taken from the examples section at https://subcheck.ioand are the results of validating a real subtitle.

6.1.1. The Report View

The report view allows the display of general metadata, such as word counts or other metrics, and allows users to gather insight into the validation results and filter them according to their needs.

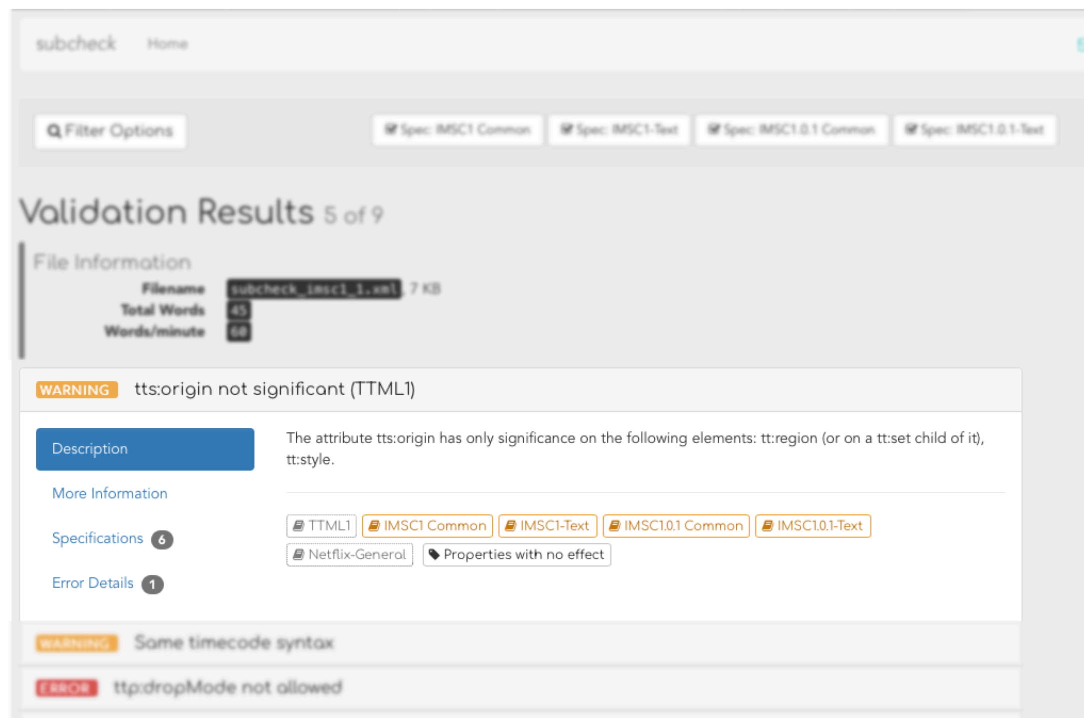
This view is built completely on the report result — so all information displayed is also available to other engines or systems.

Figure 3. The Report View



Inside the Report View, users may click on the messages to display more in-depth information.

Figure 4. Filtering



Tabs Overview

Description Tab
: The detail view starts with a general error description to add more technical context for the user. When working with different profiles within the same standard, *sub*check allows adding `Tags` to given rules that will be shown along with the general description, so users can quickly assess whether this rule's `ErrorLevel` is set differently in other profiles.

More Information Tab
: In addition to technical information, this tab will provide more general or editorially useful information and hints on why that message appeared.

Specifications Tab
: The Specifications tab points to the actual specifications this error violates. Usually entries contain a link that enables users to read the original specification for further information.

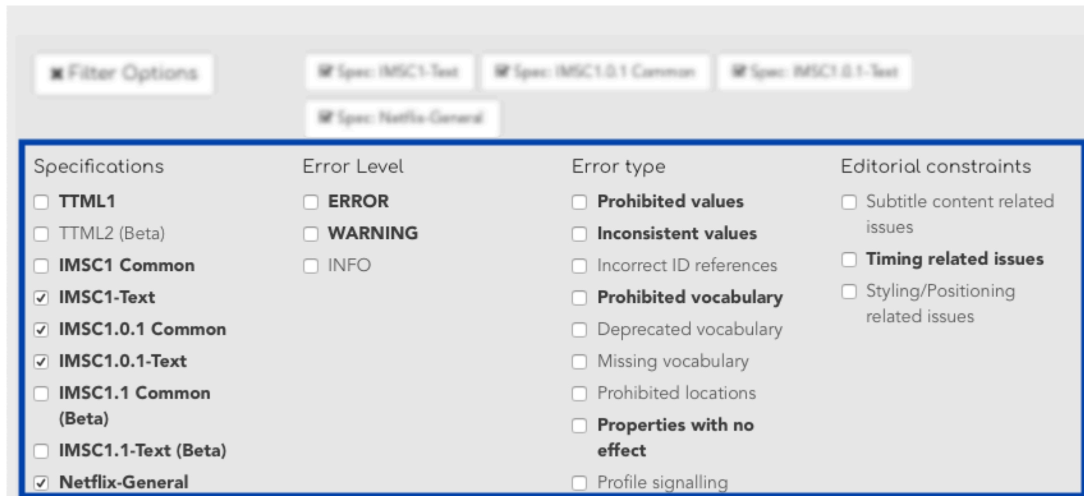
Error Details Tab
: This tab contains an `XPath`-Expression that can be copied to any capable editor or engine to pinpoint the exact error location. An HTML preview of the error is also provided, so the user gets some document context.

6.1.2. Adaptive Filtering

A *sub*check application can be used to validate documents against arbitrary schemas and allows users to provide user-defined tags that apply to a given validation message.

The screenshot below shows filtering based on a real-world subtitle validation framework:

Figure 5. Filtering



The frontend automatically adapts itself to the validation results, in a way that allows users to explore their data with these facets. The screenshot above shows, highlighted in bold, what kinds of messages are contained within the result. Users may check or uncheck these categories to display or hide specific items in the report view.

For example, some users might only be interested in messages with a `WARNING`-Level, or maybe only warnings that apply to styling-related messages.

7. Conclusion

The separation of concerns was successfully implemented.

Users can write the rules. They can update the documentation without interfering with validation or product design.

Developers can implement the validation rules. They do not have to rely on the domain user (unless there is a new rule that is a breaking change). Although the main implementation is in Schematron, other schema languages can be used as well, as long as a reported error links back to the documented rule.

Product designers can rely on the fact that, on the one hand, the same API is triggered to submit the file and filter criteria and, on the other hand, to always get back an XML document in the same format.

8. Other Aspects and Perspective

Some interesting aspects could not be covered in this paper but should be mentioned shortly:

- *sub*check was also used to validate the binary subtitle file format EBU STL. To make this possible the binary STL file was translated first into an XML representation of the binary structure. After that, the resulting XML structure was validated using Schematron rules.
- Apart from the *sub*check service, the validation engine was also implemented in the workstation-based product IMF analyser. The tool is written in C# and C++.
- Especially the integration of the W3C XML Schema validation was a challenge. It needs more work and design changes to become a generic solution.

Bibliography

[EBUTTD] TECH3380 EBU-TT-D SUBTITLING DISTRIBUTION FORMAT. W3C https://tech.ebu.ch/publications/tech3380

[ISOSCHEMA] ISO/IEC 19757-3, Information technology — Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron. 15 January 2016, ISO/IEC http://standards.iso.org/ittf/PubliclyAvailableStandards/c055982_ISO_IEC_19757-3_2016.zip

[MAFICHTE] Barbara Fichte: Strategien zur benutzerorientierten Konformitätsprüfung von XML-Dokumenten. 15 June 2014, Technische Universität München

[IMSC11] Pierre Lemieux: TTML Profiles for Internet Media Subtitles and Captions 1.1. W3C https://www.w3.org/TR/ttml-imsc1.1/

[TTML2] Glenn AdamsCyril Concolato: Timed Text Markup Language 2 (TTML2). W3C https://www.w3.org/TR/ttml2/

A. Appendix A - Subcheck artifacts with TTML examples

Example of TTML specification listing in the constraints.xml:

```
<Specifications>
  <Specification ID="spec-imsc1.1">
         <Name>
            TTML Profiles for Internet Media
            Subtitles and Captions 1.1
          </Name>
         <Acronym>IMSC1.1 Common</Acronym>
         <Version>2018-11-08</Version>
  </Specification>
  <Specification ID="spec-ttml2">
         <Name>
            Timed Text Markup Language 2
            (TTML2) CR
         </Name>
         <Acronym>TTML2</Acronym>
         <Version>2018-11-08</Version>
  </Specification>
</Specifications>
```

Example of a documented IMSC rule:

```
<Constraint ID="d1e1321">
    <ShortName>
        Extent attribute presence on region
        element
   </ShortName>
    <SpecifiedBy>
        <SpecificationReference>
            spec-imsc1-text
        </SpecificationReference>
        <SpecText>
         [The feature] #extent-region [is]
         permitted...
        </SpecText>
        <Error_Level>ERROR</Error_Level>
        <Chapter>7.4</Chapter>
        <URI>
          https://www.w3.org/TR/ttml-imsc1/#features-and-extensions-1
        </URI>
    </SpecifiedBy>
```

```
    <SpecifiedBy>
        <SpecificationReference>
            spec-imsc1.0.1-text
        </SpecificationReference>
        <SpecText>...</SpecText>
        <Error_Level>ERROR</Error_Level>
        <Chapter>7.4</Chapter>
        <URI>
            https://www.w3.org/TR/ttml-imsc1.0.1/#features-and-extensions-1
        </URI>
    </SpecifiedBy>
    <SpecifiedBy>
        <SpecificationReference>
            spec-imsc1.1-text
        </SpecificationReference>
        <SpecText>...</SpecText>
        <Error_Level>ERROR</Error_Level>
        <Chapter>8.4.2</Chapter>
        <URI>
          https://www.w3.org/TR/ttml-imsc1.1/#extent-region
        </URI>
    </SpecifiedBy>
    <ShortDescription>
        tts:extent on all regions
    </ShortDescription>
    <ShortDescriptionUser>
        The extent attribute shall be present on all region
        elements.
    </ShortDescriptionUser>
</Constraint>
```

Example of rule implementation in Schematron:

```
<sch:pattern id="attributeRequirement">
    <sch:rule
     context="/tt:tt/tt:head/tt:layout/tt:region">
      <sch:assert
       diagnostics="elementId"
       see="http://www.irt.de/subcheck/constraints/d1e1321"
       id="assert-d1e1321-1">
        The attribute tts:extent is present.
      </sch:assert>
    </sch:rule>
</sch:pattern>
```

Example of an IMSC document with an error:

```
<tt xmlns="http://www.w3.org/ns/ttml"
    ttp:profile="http://www.w3.org/ns/ttml/profile/imsc1/text"
    xmlns:ttp="http://www.w3.org/ns/ttml#parameter"
    xmlns:tts="http://www.w3.org/ns/ttml#styling"
    xml:lang="en">
    <head>
        <layout>
            <region tts:origin="10% 80%"
             xml:id="bottom"/>
        </layout>
    </head>
    <body>
        <div>
            <p region="bottom" begin="0s" end="1s">
                Hello, I am Mork from Ork.
            </p>
```

```
        </div>
    </body>
</tt>
```

Example of the SVRL Output:

```
<svrl:failed-assert
   test="attribute::tts:extent"
   see="http://www.irt.de/subcheck/constraints/d1e1321"
   location="/*:tt[namespace-uri()='http://www.w3.org/ns/ttml'][1]/..."
   subcheck:alternativeLocation="/tt/head/layout/region">
    <svrl:text>
        The attribute tts:extent is present.
    </svrl:text>
    <svrl:diagnostic-reference
     diagnostic="elementId">
       The affected 'region' element has the ID 'bottom'.
    </svrl:diagnostic-reference>
</svrl:failed-assert>
```

Example of the report view output:

```
<errorCategory>
    <constraintID>d1e1321</constraintID>
    <title>
        Extent attribute presence on region element
    </title>
    <shortUserDesc>
        The extent attribute shall be present on all
        region elements.
    </shortUserDesc>
    <specs>
        <spec>
            <name>
                TTML Profiles for Internet Media
                Subtitles and Captions 1.0 (IMSC1)-
                Text Profile, Version 2016-04-21
            </name>
            <nameAcronym>IMSC1-Text</nameAcronym>
            <text>
                [The feature] #extent-region [is]
                permitted ...
            </text>
            <errorLevel>ERROR</errorLevel>
            <section>Chapter 7.4</section>
            <uri>
                https://www.w3.org/TR/ttml-imsc1...
            </uri>
        </spec>
        <spec>
            <name>
                TTML Profiles for Internet Media
                Subtitles and Captions 1.0.1 (IMSC1)
                - Text Profile, Version 2017-07-13
            </name>
            <nameAcronym>IMSC1.0.1-Text</nameAcronym>
            <text>
                [The feature] #extent-region [is]
                permitted...
            </text>
            <errorLevel>ERROR</errorLevel>
            <section>Chapter 7.4</section>
            <uri>
```

```
                https://www.w3.org/TR/ttml-imsc1.0.1/...
            </uri>
        </spec>
        <spec>
            <name>
                TTML Profiles for Internet Media
                Subtitles and Captions 1.1 -
                Text Profile,Version 2017-10-17
            </name>
            <nameAcronym>IMSC1.1-Text (Beta)</nameAcronym>
            <text>
                The tts:extent attribute SHALL be present
                on all region elements, where it SHALL use
                px units, percentage values, or root
                container relative units.
            </text>
            <errorLevel>ERROR</errorLevel>
            <section>Chapter 8.4.2</section>
            <uri>
                https://www.w3.org/TR/ttml-imsc1.1/...
            </uri>
        </spec>
    </specs>
    <errors>
        <error>
            <messages>
                <messageMain>
                   Assertion: The attribute tts:extent is
                   present.
                   Error Information:
                   The affected 'region' element has the ID
                  'bottom'.
              </messageMain>
                <messageAssertion>
                   The attribute tts:extent is present.
                </messageAssertion>
                <messageDiagnosticsAll>
                    The affected 'region' element has the ID
                    'bottom'.
                </messageDiagnosticsAll>
            </messages>
            <locations>
                <location
                  locationType="resolvableXPATH">
                    /*:tt[namespace-uri()='http://www.w3.org/...
                </location>
                <location
                 locationType="humanXPATH">
                    /tt/head/layout/region
                </location>
            </locations>
        </error>
    </errors>
</errorCategory>
```

An Improved diff3 Format for Changes and Conflicts in Tree Structures

Robin La Fontaine

Nigel Whitaker

Abstract

There are some pieces of software, and some formats, that are de-facto standards and have been around for decades. One of these is the diff3 format for representing changes and conflicts in text documents. Diff3 works well for unstructured text documents that are divided into lines. It works surprisingly well for pretty-printed source code and similar documents. But it has frustrating limitations when used for XML or JSON or similar tree-based data formats.

Can we improve on diff3 without making it too complicated? Can the existing representation of changes and conflicts be extended to handle tree-based data? This paper seeks to answer these questions and to describe how further benefits can be enjoyed by using XML or JSON as the basis for showing conflicts and changes.

1. Introduction and Background

This paper is focused on the diff3 format rather than the diff3 executable application. In this paper, we will consider the diff3 format from GNU diffutils [1 [99]]. There are many possible outputs from diff3, but the one we are interested in is the one that provides a merged file result with conflicts marked up, i.e., the '-m' option on the command line.

The diff3 format can present information that is used in a three-way merge. It is a derivative of the two-way diff change format that uses a subset of the change markers (it does not include the ancestor information, but does use left and right angle brackets to delimit the two inputs). Many users do not invoke diff3 directly; instead, it is often invoked by a version control system such as git or mercurial when the users merge a branch, cherry-pick, rebase or change branches with working directory changes.

The format can be used for resolving changes directly, perhaps using a simple text editor, and this was a common mode of operation with early version control systems. It can also be suitable for use with a GUI to provide accept/reject changes, resulting in a new version of the file with the conflicts resolved.

In order to better understand the format itself, we will provide some background on how the diff3 tool identifies areas of conflict We will not go into any details about the limitations of using line-based comparison tools on tree-structured data, which is a subject that has been explored elsewhere and whose limitations are well known in principle if not in detail, as are a number of different ways to make a line-based comparison work better with tree-structured data, e.g., re-formatting into some canonical form.

It is possible to do a better job of comparison for XML and JSON if the comparison engine is aware of the tree structure. The issue then is how to represent the change in a way that is suitable for other systems, for example, Visual Studio Code [2 [99]], which understands the diff3 format. With some ingenuity, certain changes can be represented so that accepting or rejecting the change results in a well-formed output. However, such a representation is not always possible when, for example, start and corresponding end tags have been added or deleted, or when changes are nested.

We will propose a way that the diff3 format could be extended to handle ‘connected changes’ where the acceptance of one change requires the acceptance (or rejection) of a connected change, for example, to keep start/end tags or braces balanced. We will explore the difficulties in trying to extend it further to handle nested changes and propose a way to use XML or JSON to achieve this in a way that is more suited to those technology stacks.

2. How diff3 Delimits the Extent of Changes and Conflicts

The example is based on this paper, "A Formal Investigation of Diff3" [3 [99]]. It explains how the two two-way diffs are aligned. It is useful to understand this in order to see how it might affect tree-structured data.

The example consists of three text files with numbers on each line, A, B and the 'old' file O, as shown below:

Table 1.

A.txt	O.txt	B.txt
1 4 5 2 3 6	1 2 3 4 5 6	1 2 4 5 3 6

The way these numbers are combined into the two diffs, A+O and O+B, is shown in the table below.

Diff3 alignment across two diffs

A	O		O	B		A	O	B
1	1		1	1		1	1	1
4			2	2		4		
5			3			5		
2	2		4	4		2	2	2
3	3		5	5		3	3	
	4			3			4	4
	5		6	6			5	5
6	6							3
						6	6	6

The last three columns show how the two diffs are combined. Note that the yellow match shows where all three files align. This alignment is important because it is the data between these alignment points that are considered as units of change. Now we can look at the diff3 output using the -m option:

```
1
4
5
2
<<<<<<< A.txt
3
||||||| O.txt
3
4
5
=======
4
5
3
>>>>>>> B.txt
6
```

This output shows that the '4 5' sequence has been accepted as the only possibility between the '1' and the '2', but between the '2' and the '6' we have three possible choices, which are listed in the output. We do not want to get diverted into a

discussion about alignment algorithms, nor whether or not this is appropriate for tree-structured data. The point here is that the positions in the files at which they all three align are considered 'anchor' points, and all of the data between is considered to be a choice - and when there is some kind of conflict, the choice is left for the user to select.

There is an interesting consequence of this structure: it is not possible to have two consecutive choices without a separator that is due to a commonality between all three files, i.e., an anchor point. Although for structured data it would be natural, for example, to provide choices about attributes in a manner that allows each attribute to be chosen separately, the diff3 format dictates that two adjacent changes are seen as one choice. For structured data such as XML, it may be possible to get round this by artificially creating anchor points that are white space which is not relevant to the result. However, this is not ideal, partly because diff3 would not create such artificial anchor points and, therefore, the subsequent change to the layout of the files would not be expected by the user.

The diff3 format provides a way to delineate the three choices, though not all of them may be present. Each choice is independent of any other choice, and there is no connection between them. This independence presents a problem for tree-structured data because there is a dependency between, for example, inserting a start tag and inserting the corresponding end tag; unless these insertions are done as a single choice, the result will not be well-formed. This problem can always be overcome by duplicating some of the data, and the argument here is very similar to that presented at this conference last year regarding change to both content and structure [4 [100]]. Duplication can work well when the span of the change is small because very little data needs to be duplicated, but when the span is larger, more data needs to be duplicated and the nature of the change is lost in this duplication. In the extreme, duplication of the entire contents of an XML file will always yield a choice between well-formed fragments because each fragment is the entire file. This is correct but, of course, of little practical use.

We will look at some examples of changes to attributes where we can, with some manipulation of the data, present choices where the selection of any one of the two or three choices will provide a well-formed result.

3. Preserving Well-Formed Tree Structure in diff3

In this section, we explore the issues of preserving the well-formed structure of XML or JSON when presenting choices in diff3.

3.1. Representing XML Element Tag Change in diff3

XML tags present a problem for diff3 format in that it is, in general, not possible to ensure a well-formed result without unacceptable duplication of content. Here is an example of a change of structure.

Table 2. XML tag change

A.txt	O.txt	B.txt
This is a long paragraph where most of it has been made either bold or italic, but the rest of the paragraph remains unchanged - there is no change to the text so we do not want it duplicated. 	This is a long paragraph where most of it has been made either bold or italic, but the rest of the paragraph remains unchanged - there is no change to the text so we do not want it duplicated. 	This is a long paragraph where <italic>most of it has been made either bold or italic, but the rest of the paragraph remains unchanged - there is no change to the text so we do not want it duplicated</italic>.

This could be represented as shown below, but there is duplication of unchanged text. Such duplication is confusing because if there had been a small change, the user would have found it difficult to see.

```
<p>This is a long paragraph
where
<<<<<<< A.txt
<strong>most of it has
been made either bold or
italic, but the rest of
```

```
the paragraph remains
unchanged - there is no
change to the text so
we do not want it
duplicated</strong>
||||||| O.txt
most of it has
been made either bold or
italic, but the rest of
the paragraph remains
unchanged - there is no
change to the text so
we do not want it
duplicated
=======
<italic>most of it has
been made either bold or
italic, but the rest of
the paragraph remains
unchanged - there is no
change to the text so
we do not want it
duplicated</italic>
>>>>>>> B.txt
. </p>
```

We can improve this representation, but at the cost of some intelligence on the part of the user to make consistent choices.

```
<p>This is a long paragraph
where
<<<<<<< A.txt
<strong>
||||||| O.txt

=======
<italic>
>>>>>>> B.txt
most of it has
been made either bold or
italic, but the rest of
the paragraph remains
unchanged - there is no
change to the text so
we do not want it
duplicated
<<<<<<< A.txt
</strong>
||||||| O.txt

=======
</italic>
>>>>>>> B.txt
. </p>
```

What we really need here is some way to connect the relevant consistent choices so that if the start tag is selected, then the appropriate choice of the end is also made automatically. One simple way to achieve this would be to add a choice id into the format. In this case, we have given the three choices an id value of 42. This is shown below.

```
<p>This is a long paragraph
where
<<<<<<<42< A.txt
<strong>
||||||| O.txt
```

```
=======
<italic>
>>>>>>> B.txt
most of it has
been made either bold or
italic, but the rest of
the paragraph remains
unchanged - there is no
change to the text so
we do not want it
duplicated
<<<<<<<42< A.txt
</strong>
||||||| O.txt

=======
</italic>
>>>>>>> B.txt
. </p>
```

There are many ways this connection could be achieved syntactically; this is just one. The rules here would be:

1. A conflict may be labelled with an id.
2. For any labelled conflict, there must be at least one other labelled conflict with the same id value.
3. The selection of a choice within a conflict with an id automatically results in the selection of the corresponding choice, i.e., the choice with the same source file, within conflicts with the same id.

Putting the numbers is not a big change to the format but would make a significant difference to the ease of use of diff3 format for structured data,.

3.2. Representing XML Attribute Change in diff3

XML attributes present a particular challenge for diff3 format. Here is an example of a change of value for an attribute.

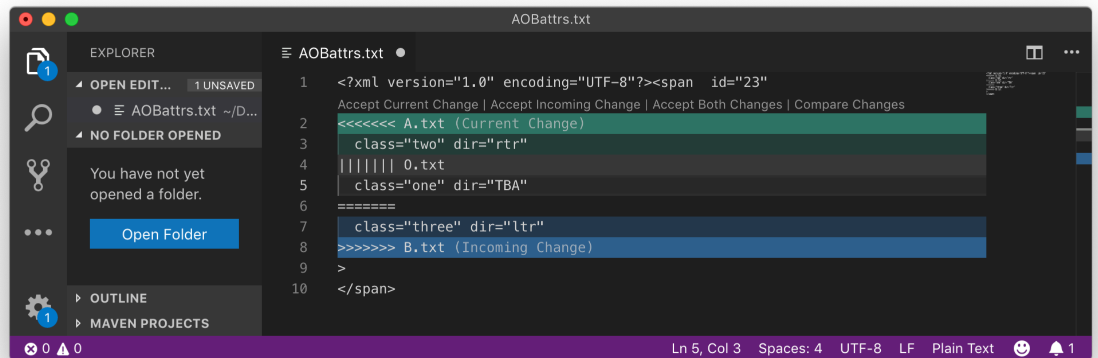
Table 3. XML attribute value change

A.txt	O.txt	B.txt
<span class="two" dir="rtr" id="23">	<span class="one" id="23" dir="TBA">	<span id="23" class="three" dir="ltr">

This could be represented as shown below. Note here that we are not showing the result of running 'diff3 -m' but rather we have run an XML-aware comparison yielding results that we want to express in the diff3 format.

```
<span id="23"
<<<<<<< A.txt
class="two" dir="rtr"
||||||| O.txt
class="one" dir="TBA"
=======
class="three" dir="ltr"
>>>>>>> B.txt
>
```

Figure 1. Attribute example in Visual Studio code



In Figure 1 [93], we see how this can be displayed and managed in Microsoft Visual Studio code.

The above will produce syntactically correct results, though it is not ideal because it would be more natural to choose the attributes separately rather than as a pair. This separation can be achieved by inserting additional white space so that we get two choices as shown below.

```
<span id="23"
<<<<<<< A.txt
class="two"
||||||| O.txt
class="one"
=======
class="three"
>>>>>>> B.txt

<<<<<<< A.txt
dir="rtr"
||||||| O.txt
dir="TBA"
=======
dir="ltr"
>>>>>>> B.txt
>
```

There is another representation that takes the common attribute name out of the choice, but it may be less easy for a user to see what is happening. This representation is shown below.

```
<span class=
<<<<<<< A.txt
"two"
||||||| O.txt
"one"
=======
"three"
>>>>>>> B.txt
 dir=
<<<<<<< A.txt
"rtr"
||||||| O.txt
"TBA"
=======
"ltr"
>>>>>>> B.txt
```

```
>
```

3.3. Representing JSON Structure Change in diff3

For JSON, the issue of handling curly braces (for objects) and square brackets (for arrays) is similar to the issue of handling XML start and end tags. Again, some representation of connected change is needed to maintain syntactic correctness.

Object members and array members are comma separated, and this syntax is tricky to get right in some situations. The syntax is shown below.

```
object = begin-object [ member *( value-separator member ) ]
               end-object
array = begin-array [ value *( value-separator value ) ] end-array

These are the six structural characters:
  begin-array = ws %x5B ws ; [ left square bracket
  begin-object = ws %x7B ws ; { left curly bracket
  end-array = ws %x5D ws ; ] right square bracket
  end-object = ws %x7D ws ; } right curly bracket
  name-separator = ws %x3A ws ; : colon
  value-separator = ws %x2C ws ; , comma

Insignificant whitespace is allowed before or after any of the six
structural characters.
ws = *(
       %x20 / ; Space
       %x09 / ; Horizontal tab
       %x0A / ; Line feed or New line
       %x0D ) ; Carriage return
```

Here is an example of a change to an array of strings.

Table 4. JSON structural change

A.txt	O.txt	B.txt
[[12,13,14],20,21,22]	[12,13,14,20,21,22]	[[12,13,14,20,21,22]]

This could be represented as shown below.

```
[
<<<<<<<42< A.txt
[
||||||| O.txt

=======

>>>>>>> B.txt

<<<<<<<61< A.txt

||||||| O.txt

=======
[
>>>>>>> B.txt
12,13,14
<<<<<<<42< A.txt
]
||||||| O.txt

=======

>>>>>>> B.txt
```

```
,20,21,22
<<<<<<<61< A.txt

||||||| O.txt

=======
]
>>>>>>> B.txt
]
```

The above will produce syntactically correct results, though it is not intuitive and requires careful allocation of id values for the conflicts to ensure correct behaviour. Note that the '[' in A has to be a separate conflict from the '[' in B, although they are at the same position in the array. Note that it could be argued that these changes are not conflicts, but this does not matter here; the point is that if we do want to represent them as choices for a user to select, then we are able to do so.

3.4. Representing JSON Separator Change in diff3

The problem with separators is that they cannot consistently be associated with either the start or the end of each item (member for object and value for array) because if there is only one item then no separator is needed. Therefore, maintaining correct syntax when items are added or deleted is not trivial. As mentioned above, the diff3 format does not allow consecutive choices without 'anchor' data between, so it is necessary to group consecutive items that may be added or deleted into one choice. This apparent restriction does lead to a greater likelihood of the syntax of each choice being consistent.

Here is an example of a change to an array of strings.

Table 5. JSON array value change

A.txt	O.txt	B.txt
`["one", "two"]`	`["one"]`	`["three", "four"]`

This could be represented as shown below. Note here that we are not showing the result of running 'diff3 -m' but rather we have run an JSON aware comparison yielding results that we want to express in the diff3 format.

```
[
<<<<<<< A.txt
"one", "two"
||||||| O.txt
"one"
=======
"three", "four"
>>>>>>> B.txt
]
```

The above will produce syntactically correct results, though it is not ideal because it would be more natural to choose the values separately rather than as a complete list. This separation can be achieved by inserting additional white space so that we get two choices as shown below.

```
[
<<<<<<< A.txt
"one"
||||||| O.txt
"one"
=======

>>>>>>> B.txt

<<<<<<< A.txt
, "two"
||||||| O.txt

=======

>>>>>>> B.txt
```

```
<<<<<<< A.txt
||||||| O.txt
=======
, "three", "four"
>>>>>>> B.txt
]
```

However, this representation can lead to syntax errors because if the "one" is rejected by accepting the B.txt choice in the first conflict, then we do not need a comma before the next item. Unfortunately, we cannot get round that using a connected choice. The problem here has to do with a combination of choices rather than one choice. We can just be pleased that XML attributes are not comma separated!

4. diff3 Format as XML or JSON

An obvious question about diff3, when we are looking at XML and JSON, is whether or not we would get a significantly better result if we used XML or JSON instead of the fairly basic format of diff3. The table below shows an example in diff3 and the corresponding file in XML and JSON using a very simple syntax in each case. The purpose here is just to explore whether or not it makes sense to do this.

Table 6. diff3 format in XML or JSON

<table>
<tr><th>diff3</th><th>XML</th><th>JSON</th></tr>
<tr>
<td>1
4
5
2
<<<<<<< A.txt
3
||||||| O.txt
3
4
5
=======
4
5
3
>>>>>>> B.txt
6</td>
<td><d:diff3>
1
4
5
2
<d:change>
    <d:content origin="A.txt">
3
</d:content>
    <d:content origin="O.txt">
3
4
5
</d:content>
    <d:content origin="B.txt">
4
5
3
</d:content></d:change>
6
</d:diff3></td>
<td>{
 "diff3": [
  "1",
  "4",
  "5",
  "2",
  {
   "change": {
    "A.txt": ["3"],
    "O.txt": [
     "3",
     "4",
     "5"
    ],
    "B.txt": [
     "4",
     "5",
     "3"
    ]
   }
  },
  "6"
 ]
}</td>
</tr>
</table>

For JSON, we have represented the sequence of lines as an array of strings, where each line is a string and a change is an object where each member name is the name of the original file. We could have concatenated the lines with a '\n' delimiter, but this would have been very difficult to read.

This example shows that JSON changes the look and feel significantly due to the way it represents strings. XML is similar to the original, though some detail is left out here, for example, `xml:space="preserve"` or `<![CDATA[` to preserve the formatting. If the original data is XML, then representing the changes in XML in this way would be very confusing and it would be better to embed the changes within the original XML, assuming the original was well-formed.

The addition of the id (to represent connected changes) would be very simple in XML as an attribute, but a little harder in JSON because it would mean adding another member to the change object. The table below compares some of the characteristics of the three formats, where we use an informal score of three stars for good, two stars for OK and one star for poor.

Table 7. Characteristics of diff3, XML and JSON

Characteristic	diff3	XML	JSON	Comment
No processing needed for unchanged file	***	**	*	
Preserve line structure	***	***	**	JSON needs strings or \n
Good for text editor (by hand)	***	*	*	
Connected changes	**	***	**	
Nested changes	*	***	**	
Changes within a line	*	***	**	
Show all resolved merges	*	***	**	
Show changes to JSON data	**	**	***	
Show changes to XML data	*	***	*	

The table does show some potential advantages of having an XML representation of diff3, especially for automated processing. For showing changes to well-formed XML in XML this might require some care to preserve comments, processing instructions and the first line declaration/prolog. Attribute changes could also not be handled as text so again would need some further design thought. One approach would be to treat the XML source file as text and enclose it in CDATA sections. It is likely that embedding the changes in a well-formed XML source would require a different approach to simply using XML to show changes in a text file. Similar issues would occur for showing changes to JSON in JSON.

This proposal is not intended as an alternative to diff3 and it is clear that there would be issues to resolve if JSON or XML were used. XML does look more appropriate, but it lacks one desirable characteristic of diff3: no processing is needed for an unchanged file (or one with no conflicts).

5. Nested Changes

Given an XML or JSON representation, we can go one step further and use the fact that the representation is hierarchical to support hierarchical or 'nested' change. A nested change is a change in one branch that modifies something that has been removed in another branch.

We will look at an XML example, showing nested changes.

Table 8. XML nested data example

A.xml	O.xml	B.xml
<author> <personname> <firstname>Nigel </firstname> <surname>Whitaker </surname> </personname> <address> <phone>+44 1684 532141 </phone> <street>Geraldine Road </street> <city>Malvern</city> <country>UK</country> <postcode>WR14 3SZ </postcode> </address> </author>	<author> <personname> <firstname>Nigel </firstname> <surname>Whitaker </surname> </personname> <address> <street>Geraldine Road </street> <city>Malvern</city> <country>UK</country> <postcode>WR14 3SZ </postcode> </address> </author>	<author> <personname> <firstname>Nigel </firstname> <surname>Whitaker </surname> </personname> </author>

In the above example, one branch, `B.xml`, has deleted the `address` sub-tree, which the other branch has modified with an added `phone` number.

Let us now consider how this could be represented using the proposed XML format presented in the previous section:

```
<d:diff3>
<author>
```

```
  <personname>
    <firstname>Nigel</firstname>
    <surname>Whitaker</surname>
  </personname>
  <d:change>
    <d:content origin="A.xml">
      <address>
        <phone>+44 1684 532141</phone>
        <street>Geraldine Road</street>
        <city>Malvern</city>
        <country>UK</country>
        <postcode>WR14 3SZ</postcode>
      </address>
    </d:content>
    <d:content origin="O.xml">
      <address>
        <street>Geraldine Road</street>
        <city>Malvern</city>
        <country>UK</country>
        <postcode>WR14 3SZ</postcode>
      </address>
    </d:content>
    <d:content origin="B.xml"/>
  </d:change>
</author>
</d:diff3>
```

Here we can see the deletion of the address in `B.xml`, and if we carefully look at `A.xml` and `O.xml`, we can work out that a phone child element has been added. But is there a better representation we can use? Given we are now using a representation that follows the tree structure, we can also make use of this structure in the result. Here is an alternative result, where we allow change to nest:

```
<d:diff3>
<author>
  <personname>
    <firstname>Nigel</firstname>
    <surname>Whitaker</surname>
  </personname>
  <d:change>
    <d:content origin="A.xml, O.xml">
      <address>
        <d:change>
          <d:content origin="A.xml">
            <phone>+44 1684 532141</phone>
          </d:content>
          <d:content origin="O.xml"/>
        </d:change>
        <street>Geraldine Road</street>
        <city>Malvern</city>
        <country>UK</country>
        <postcode>WR14 3SZ</postcode>
      </address>
    </d:content>
    <d:content origin="B.xml"/>
  </d:change>
</author>
</d:diff3>
```

Here we can see that by allowing nested change and making some small adjustments to the format to allow multiple versions to be specified in origin attributes, we can avoid the repetition and make it easier for a human to understand. However, we have moved further from the simple diff3 representation in this step. Rather than choose one of two or three possibilities at each step, we now need to understand reuse of content, and a more complex format is used for the origin attributes.

As well as being more compact and allowing reuse, there is a further benefit: in some cases, nested changes can be ignored. Suppose we decided to accept the change made in `B.xml`, the deletion of the address. In this case, we would take the `B.xml` content, i.e., nothing, and immediately delete the other `d:content` alternatives at that level of the tree. We do not need to consider the nested change related to the phone element when we choose the outer `B.xml` alternative.

It is also possible to prove that for a three-way merge process, as used by diff3, at most two levels of nested change/content structure is required. This can be generalized so that for n-way merge algorithms (akin to the idea of 'octopus merge' used in git), a maximum of n-1 levels of nested change are required.

We have not explored how the original diff3 representation could be enhanced to handle nested change. It could be done, but it is more natural in the context of an XML representation of change.

6. Conclusions

In this paper, we have explored whether we can integrate more modern structure-aware comparison tools with the existing diff3 format so that there is minimal change for users. We have shown that by laying out comparison results for structured representations such as XML or JSON, we can make them easier to process and more likely to provide well-formed or valid results. We have shown that there are limitations, in particular the representation of connected changes, where some more intelligence in the diff format is needed to ensure the result is well-formed. Nested changes can also be represented with amendments to the diff format, but this is more complicated and is likely to be easier using XML rather than a variant of diff3.

Even if these things are possible, that does not necessarily mean we should go down this route. It is worth considering some of the history and how we got here. Early version control systems were in use with 24 line, 80 column VDUs and with editing tools such as vi and emacs. In those days, developers intimately understood the representation and manipulated it directly. We are now used to using IDEs that directly support version control operations in their graphical user interfaces. In many cases, these interfaces hide the change markers that we have been discussing and instead present the user with side-by-side alternatives and GUI control buttons to resolve differences. We could consider the display of the diff3 style change-markers akin to the concept of 'tag display' modes in word-processors and XML editors. In many of these systems, either it is impossible to see any underlying markup or it is a feature for advanced (or perhaps 'older'?) users that needs to be explicitly turned on, with the growing trend for the default being to hide the markup from the user. Is there a similar trend with change and conflict markers? This implies that the actual syntax used to represent the changes and conflicts is less important than it used to be, and there is less need to try to preserve it.

In our recent paper [5 [100]], we identified some issues in version control systems that caused inconsistency and confusion to users. One solution to those issues relies on separating the merge driver from subsequent conflict resolution tools or 'merge tools'. In pursuit of the best way forward, we have further explored these possibilities and we have implemented the layout approaches discussed earlier.

We have shown that improvements to the representation of change for structured data is possible and desirable. Changing the existing diff3 format is awkward and limited, so it might be better to move directly to a markup representation using XML because the text will not be directly edited by users and mature tools are available to process the XML. Arguably it would be simpler to avoid these issues and present users with a merge user interface that understood structured content and provided operations which preserved the well-formed nature or validity directly. However, the value of a standard format for such conflicts and changes is that the merge tool is primarily a GUI and the user can choose the merge algorithm and the merge tool independently.

We have presented this paper to explore these ideas, but we are not suggesting that the best approach is extending or enhancing the current diff3 representation. An XML alternative to diff3 would have some advantages and should be explored as a longer-term improvement for representing and processing conflicts and changes in structured data.

References

[1] GNU Diffutils. https://www.gnu.org/software/diffutils

[2] Visual Studio Code User Guide: Using Version Control in VS Code. https://code.visualstudio.com/docs/editor/versioncontrol

[3] S. Khanna, K. Kunal, B.C. Pierce: A Formal Investigation of Diff3 in Arvind V., Prasad S. (eds) FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science. FSTTCS 2007. Lecture Notes in Computer Science, vol 4855. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007%2F978-3-540-77050-3_40

[4] Robin La Fontaine: When Overlapping XML Meets Changing XML Does Confusion Reign? in Markup UK 2018 Proceedings. https://markupuk.org/2018/Markup-UK-2018-proceedings.pdf#page=153

[5] Robin La Fontaine and Nigel Whitaker: Merge and Graft: Two Twins That Need To Grow Apart in XML Prague 2019 Conference Proceedings. February 7-9, 2019. http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf#page=175

<Angle-brackets/> on the Branch Line

John Lumley, jωL Research,

Abstract

As a retirement 'hobby', somewhat removed from the computing *milieu*, the author has started building a model railway in his garden. Surveying the extant tools for designing such layouts and finding them "not quite right", he started building a design tool himself, using the familiar technologies of XSLT3 and SVG executing in a browser, employing Saxon-JS as the processing platform. By adding animations, the tool expanded beyond simple design to in effect become an active model train system. The results of this were demonstrated, with some success, at Markup UK in 2018. This paper describes the design of this tool in some detail, as well as possible developments since that demonstration.

1. Introduction

The author decided to take up a retirement "hobby" as a change from wrestling with programmatic complexities. Having chosen to build a garden railway, having been trained as an engineer and having read some of the sage advice from those already "in the scene", it was clear that the layout would need some careful design. Issues such as maximum gradients, minimum turn radii and *loading gauge* clearances required a clear and calculated design. Naturally there are CAD tools specifically targetted at model railways, but equally well, I found none of them to be *just right*.

So, having spent many years developing software, and in recent times being deeply immersed in XML technologies, particularly XSLT3.0[XSLT] and SVG[SVG], I decided to build a specific design tool with these technologies. Given Saxon-JS[Saxon-JS] as the XSLT execution engine, the tool was run through a browser connecting to a *localhost* web server.

The main design used an XML definition of garden "background" and the possible layouts, and at first calculated all the resulting geometry, producing both a tabular summary and a set of SVG graphic elements that could be displayed on a grid. This permitted for example interferences between tracks and garden elements (e.g. bushes) to be examined. Simple XHTML controls were added to allow various aspects of the display to be altered dynamically, using Saxon-JS's interactive modes (e.g. `ixsl:on-change`) to alter style or class properties of parts of the XHTML/SVG DOM tree. Textual styling (fonts, colours. etc.) were defined in a set of CSS stylesheets.

Once a simple system was operational, the "picture" was enhanced, both by supporting an isometric view of the garden/layout, but also more "realistic" graphics for the track and other aspects.

A little experimentation showed that the animation facilities present in SVG should allow objects to move around the paths of the track. A simple facility was added to enable "block" objects to be run, moving from section to section under controllable and alterable speeds. Simple click interaction allowed the points to be changed, so the path of these blocks could be altered whilst they were still running.

The model for these "locomotives" was improved to support a three-dimensional definition consisting of a number of orthogonal rectangular blocks and cylinders, from which an isometric SVG view of a simple locomotive could be displayed. This would then be animated to follow the path of the current track section, with tangential rotation to "point forward" and with suitable rotation animations on the wheels. Simple sound effects (running sound, whistles etc.) were added to the design.

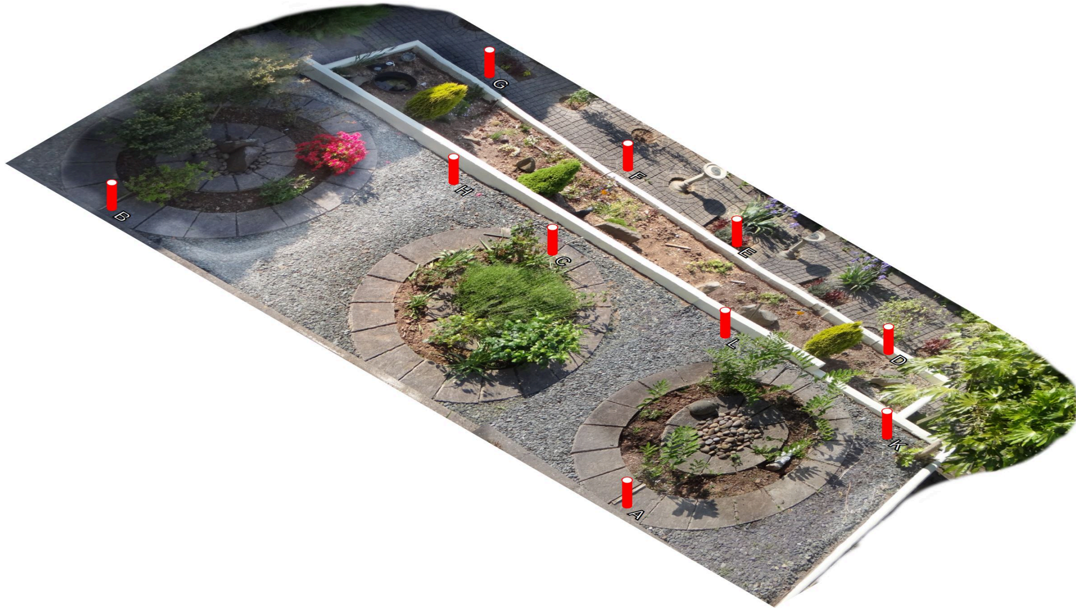
Finally, this system was demonstrated at MarkupUK 2018 in the DemoJam session, with some success.

In this paper I describe the deign and implementation details of the system that was demonstrated, and outline some additional possible developments. In conclusion I discuss how suitable the combination of XSLT3, SVG and Saxon-JS has been to tackling this design task.

2. Overall Design

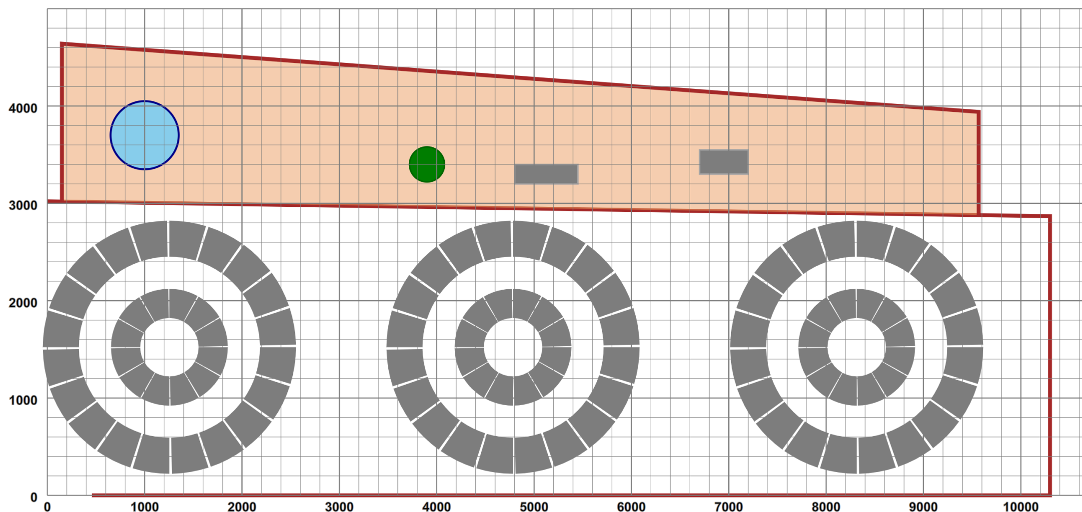
The system designed is of course influenced by the external factors of the garden itself and the track components from which the railway will be built. The garden area chosen is approximately 10m x 4m on two levels:

Figure 1. The Garden for the Railway



Both levels are substantially flat[1] with a step of about 250mm between, so it should be possible to climb a connecting embankment. (Generally gradients should be less than 1 in 40 and certainly no more than 1 in 25). The area was surveyed (marked by the red survey points shown above) and a simplified plan of the garden drawn up:

Figure 2. The garden plan



I decided that the railway would be built at SM32 gauge/scale, also known as 16mm. The track has a gauge of 32mm, and is taken to represent a 2ft narrow-gauge line[2] so the scale is 16mm to the foot or 1:19. As such, models of narrow-gauge locomotives are large enough to be totally steam-powered. The tracks themselves would not carry electrical power — all engines would be self-powered, and remotely controlled. The commercially available track had a small set of points of different tightness and flexible track sections of some 900mm length. This meant that apart from the fixed-design points, the rest of the track could be "freeform", subject of course to a recommended minimum turn radius, which whilst being dependent upon locomotive wheelbase, would be about 900-1000 mm.

The original design consists of five major sections:

[1]Only when laying out the track bed did it become apparent that several elevation changes O(50mm) existed on the upper level.
[2]Many of the UK's "little trains", such as the Ffestiniog and the Talyllyn, run on 1' 11½" gauge track

- A declarative description, as an XML structure, of the design environment, consisting of background components (e.g. pictures of the garden and schematics of fixed sections such as walls, paving and plants) and a series of layouts. A layout is described as a sequence of (mainline) track sections of straights, curves and points, each represented by an XML element describing length, radius and/or turn angle. Branch lines are children sequences of a `point` element. Where necessary track connections between leaves of the tree are joined to make a complete graph thrrough named link declarations.

- A geometry computational engine, written in XSLT3, which calculates the position and orientation of each track section, and produces a map of the layout, keyed by section 'name', each entry describing both the track segments of the section and the two-way connectivity between section ends.

- A graphical display of the design as an SVG tree. Background elements are generated as SVG groups from the enviroment description. Track components are generated from unit descriptions and positioned with *use* instructions. Within this, some components which can differ in display dependent upon state, such as points, are represented by several views, each classed separately. The overall display can be subject to transform, most noteably an isometric one. Textural styling and initial visibility is defined in a series of CSS stylesheets.

- An XSLT3 stylesheet, using Saxon-JS extensions, and invoked from an outer XHTML document, which populates the XHTML with a series of interactive controls, and generates the detailed layout internal structures and SVG graphics to be embedded in the web page. Templates respond to interaction, such as button state changes, or clicking on points levers, altering the local CSS state of other components and controls.

- Adding "railway engines" as SVG objects, which are presented in both plan and isometric views from a simple "block-and-cylinder" model. An event-based system animates these to run along tangential paths of the track sections, using SVG animation facilities. Speed and direction of travel can be controlled interactively for multiple engines. Events are generated at the conclusion of animations, and are caught by templates that consult the layout map to determine the next sector to enter, then calculate the necessary animation duration, given length and speed, and start up the path-following animation. Speed change involves stopping a current animation, recalculating duration for the remaining section path and restarting an new animation partway through. Issues on collision detection ("train crashes") will be discussed.

As far as the software mix is concerned, the top-level XHTML document contains some constant background components and `div` containers which will be populated, a `script` element containing a very small set of global JavaScript functions, for primary control of animations and mapping from screen to SVG co-ordinates, and an invocation of Saxon-JS with a precompiled program from an XSLT source of some 20 files and perhaps some 3000 source lines. This program takes as input a file containingg definitions of the garden, possible layouts and locomotives. Textural styling is supported by a set of associated (static) CSS files.

There are a number of (Javascript) libraries for supporting SVG effects and animation, and pretty much all the written guides to "advanced" SVG use a combination of some of these, but I wanted to explore how much could be done almost entirely in XSLT3.0. All the programming is limited to XDM data types, XHTML, SVG, CSS and XSLT3.0 with Saxon-JS interaction extensions, with a minimum of (perhaps a dozen) globally defined small JavaScript functions, mostly to invoke,query and stop SVG animations.

3. Layout Topology and Geometry

The original motivation was as a tool to design my planned garden railway, in terms of a connected set of track components that satisfied the requirements of i) being constructed from obtainable parts and ii) lay within the limits of bend curvature and track gradient that were recommended for such railways. For the present, given the flat nature of the garden, apart from the step between sections, vertical gradients have been ignored — how they could be added is discussed later.

I considered attempting a "drag and drop" style of interaction, but decided against this, especially as all straights and curves could be "freeform" so a small set of track parts wasn't really appropriate. The starting point was a choosing an XML representation that focussed on continuous sequences of track components, describing the "main line", implemented as a sequence of elements, such as:

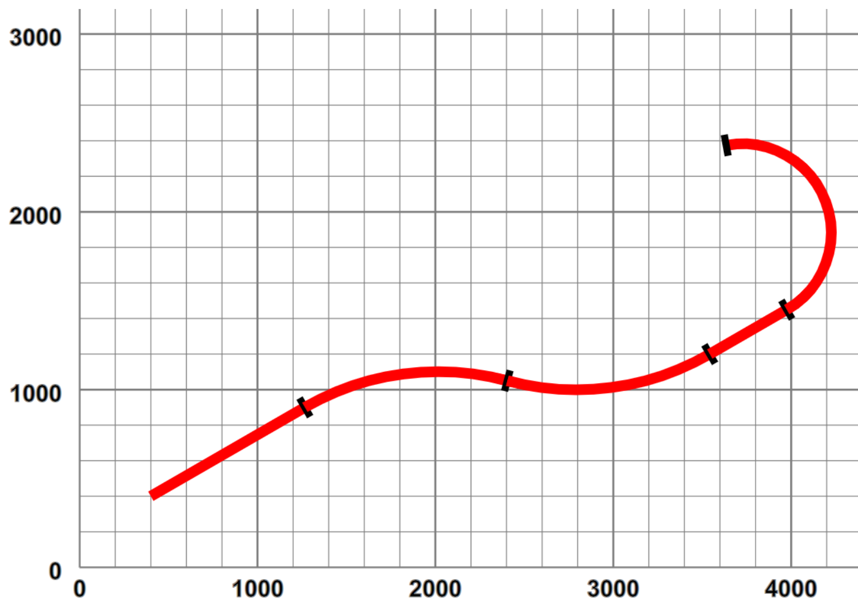
Example 1. A simple layout

```
<layout name="simple">
   <start x="400" y="400" orient="30"/>
   <straight name="section1" length="1000"/>
```

```
    <curve r="1500" angle="-45"/>
    <curve r="1500" angle="45"/>
    <straight length="500"/>
    <curve r="500" length="1400"/>
</layout>
```

which defines a layout *simple* that contains one section *section1*. This starts at the point (400,400) with an orientation of +30° from the positive X axis. The first section is a 1000 long[3] straight, preserving orientation, followed by a circular arc curve, of radius 1500, turning left though a positive angle of 45°, followed by a similar right turn, a short straight and a tighter left-hand bend defined by radius and curve length, rather than angle. When plotted out this section looks like:

Figure 3. Simple layout - pictorially



Circular arcs were chosen as the only curve representation as i) they support a design method of "turn this tightly for x degrees", ii) they are supported directly in SVG and iii) their geometry is simple to calculate. Polynomial splines could have been used, but they are difficult to define in terms of curve length. In real railway engineering, curves are defined by *Cornu spirals* - where the curvature (1/*radius*) is a piecewise linear function of arc length — lateral (centripetal) acceleration increases at a uniform rate as a train moves along such a curve at constant speed. SVG alas does not support such curves.

Layouts that have such a simple topology (a single contiguous section) tend to be somewhat boring. Alterative routes involve switching between different sections joined by *points*[4]. In our layout definition a point is represented as an element, *whose child is the "branch line"*:

Example 2. A simple branch line

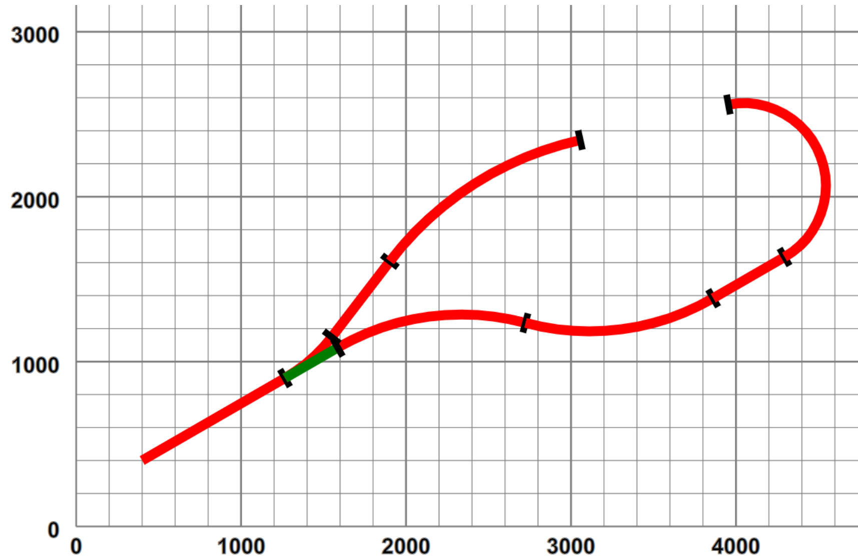
```
<layout name="simplePoint" start="section1">
    <start x="400" y="400" orient="30"/>
    <straight name="section1" length="1000"/>
    <point id="P1" radius="small" turn="left">
        <spur>
            <straight name="branch1" length="580"/>
            <curve r="2000" angle="-40"/>
        </spur>
    </point>
    <curve name="section2" r="1500" angle="-45"/>
    <curve r="1500" angle="45"/>
    <straight length="500"/>
    <curve r="500" length="1400"/>
```

[3]Any consistent distance units could be used of course, but for this case it's simplest to use millimetres.
[4]In American terminology *turnouts*.

```
</layout>
```

The branch line itself is defined by a `spur` element, whose children define a set of sections. The point defines its type, in this case a small radius point and its handedness — here the branch turns off to the left. This layout looks like:

Figure 4. Simple branch line - pictorially



The point obviously has two possible paths, one straight on, the *not-set* track, shown in green, and the turning branch, the *set* track. The layout now consists of three sections, *section1* leading up to the point *P1*, followed by *section2* as the mainline and *branch1* on the branch.

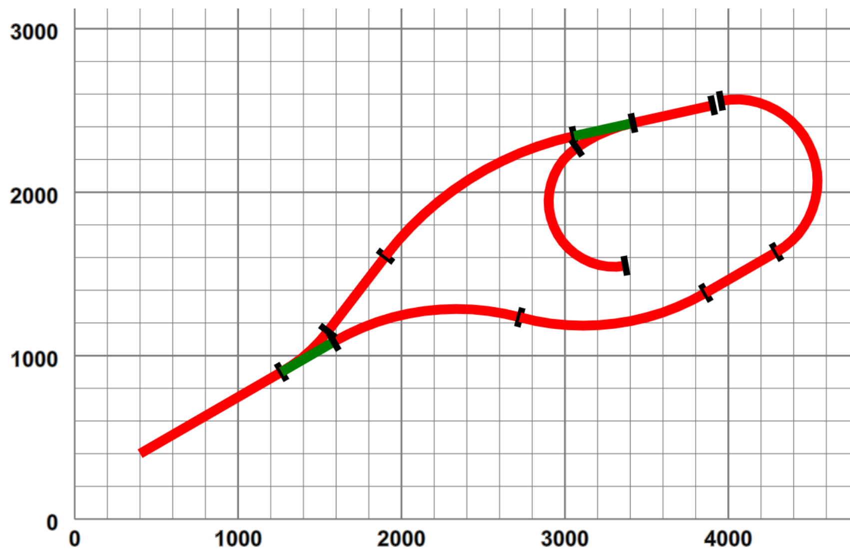
This “tree” representation can obviously be extended, such as adding a point on the branch line, with a sub-branch line such as:

Example 3. A layout with two points

```
<layout name="twoPoints" start="section1">
   <start x="400" y="400" orient="30"/>
   <straight name="section1" length="1000"/>
   <point id="P1" radius="small" turn="left">
      <spur>
         <straight name="branch1" length="580"/>
         <curve r="2000" angle="-40"/>
         <point id="P2" radius="small" dir="trailing" turn="left">
            <spur>
               <curve r="400" angle="155"/>
            </spur>
         </point>
         <straight length="500"/>
      </spur>
   </point>
   <curve name="section2" r="1500" angle="-45"/>
   <curve r="1500" angle="45"/>
   <straight length="500"/>
   <curve r="500" length="1400"/>
</layout>
```

which looks like:

Figure 5. Two points pictorially



Observant readers will note that the new point has been added in technically a *trailing* condition, i.e. proceeding from the start it is only possible to enter the siding in reverse[5]. This leads us on to considerations of representing the layout *topology*.

3.1. Representing the topology

If we want to use a layout for any purpose other than design (such as interactive animation), we don't just need the geometry of the layout: we also need to represent the topology — which sections are joined when points are in a given state? If a train leaves one section, which is the one it will enter, if any? To do this we represent contiguous sections of track and points as components with two or three *ports*:

face
: The port which faces against an oncoming vehicle in normal travel, i.e. trains usually start from the face port. For *points* this is the entry from which the exit track (*trail* or *spur*) depends upon the state of the point.

trail
: The port from which a vehicle emerges in normal travel, i.e. trains usually end a section leaving the trail port. For points entered in the normal switched direction this is the exit when the point is *not set*.

spur
: Only defined for points, the exit port when the point has been *set* [6].

Using these definitions we can describe the topological relations between component sections in a simple map:

Figure 6. Topology of a two-point layout

id	type	down	parts	length	face	trail	spur
branch1	section	false	2	1,976	P1.spur	P2.trail	
branch2	section	true	1	1,082	P2.spur		
branch3	section	false	1	500	P2.face		
section1	section	false	1	1,000		P1.face	
section2	section	false	4	4,256	P1.trail		
P1	points	false	1	369	section1.trail	section2.face	branch1.face
P2	points	false	1	369	branch3.face	branch1.trail	branch2.face

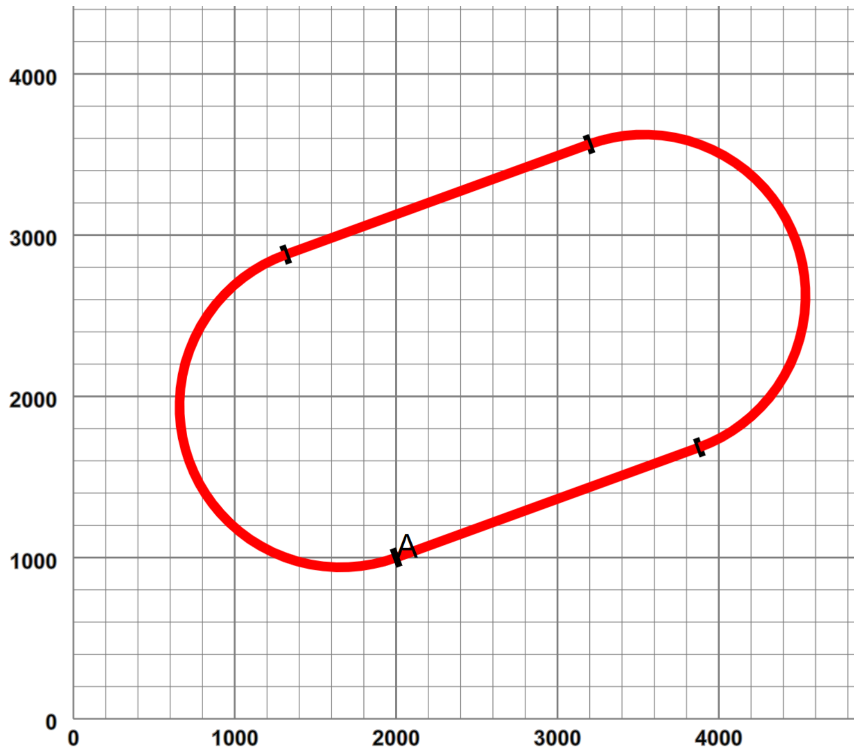
This map has an entry for each component describing its type and its port connections in terms of a component/port pair to which that port attaches. Note however that *branch2* (the “backward” spur from point *P2*) is labelled `down=false`.

[5] Early railway practice only used trailing points on higher-speed main lines, to reduce risk of derailment from partially opened points.
[6] In theory an engine entering a set of points from the *trail* or *spur* direction, when the points are set *against* that direction, i.e. when *set* from *trail* or *not set* from *spur*, may be able to “force” an automatic points switch, but this is not recommended practice.

This means that the "main" direction (i.e. proceeding from *P2* along *branch2*) of that section of track is in a reversed sense to the rest of the layout — the importance of this will become apparent later.

Thus far we have a layout that has no loops or paths of multiple connection, and whilst totally representable by a tree is not completely useful, especially if one wants to leave a train running around the layout indefinitely. Suppose we have a simple oval loop:

Figure 7. An oval becomes a loop



which starts at 2000,1000, and loops back through two straights and two curves to an end point co-incident in position *and orientation* with the start. To "close the loop", we have to convert our tree to a graph, in this case with "self-pointers" by adding a specific link directive

Example 4. Describing a graph linkage

```
<layout name="oval" start="A">
    <start x="2000" y="1000" orient="20"/>
    <straight name="A" length="2000"/>
    <curve r="1000" angle="180"/>
    <straight length="2000"/>
    <curve r="1000" angle="180"/>
    <link>A.trail A.face</link>
</layout>
```

id	type	down	parts	length	face	trail	spur
A	section	false	4	10,283	A.trail	A.face	

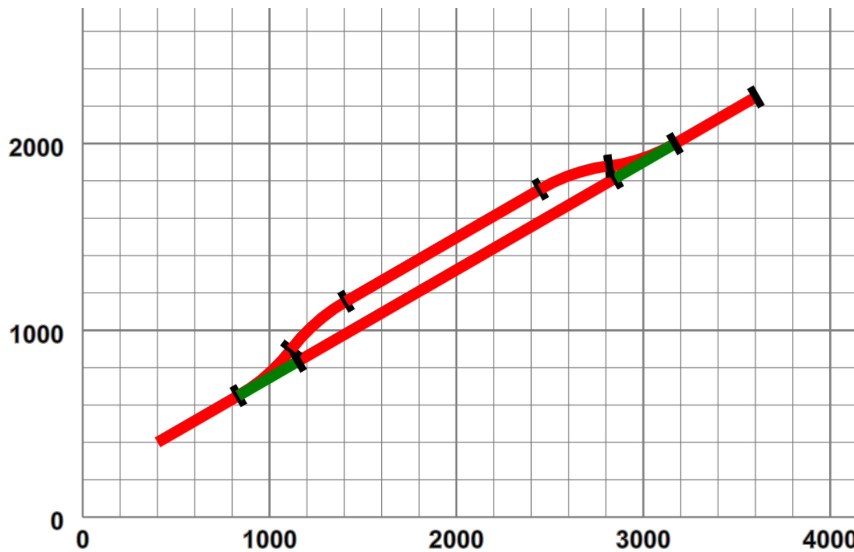
Now a vehicle finishing at *A.trail* can proceed happily into *A* again through *A.face* and similarly in a reverse direction. Of course in this case we could infer from the geometrical co-incidence that such a link may be required, but sometimes the geometry isn't quite accurate enough. Here is a passing loop:

Example 5. A passing loop

```
<layout name="passingLoop" start="main-line1">
    <start x="400" y="400" orient="30"/>
    <straight name="main-line1" length="500"/>
    <point id="passing1" radius="small" turn="left">
        <spur>
           <curve  name="passing-loop" r="1000" angle="-22.5"/>
           <straight length="1200"/>
           <curve r="1000" angle="-22.5"/>
        </spur>
    </point>
    <straight name="main-line2" length="1960"/>
    <point id="passing2" radius="small" dir="trailing" turn="right"/>
    <straight name="main-line3" length="500"/>
    <link>passing-loop.trail passing2.spur</link>
</layout>
```

Where now we have specifically linked the passing loop component onto the trailing point spur:

Figure 8. Passing loop graphically and topologically



id	type	down	parts	length	face	trail	spur
main-line1	section	false	1	500		passing1.face	
main-line2	section	false	1	1,960	passing1.trail	passing2.trail	
main-line3	section	false	1	500	passing2.face		
passing1	points	false	1	369	main-line1.trail	main-line2.face	passing-loop.face
passing2	points	false	1	369	main-line3.face	main-line2.trail	passing-loop.trail
passing-loop	section	false	3	1,985	passing1.spur	passing2.spur	

But linking isn't quite as straightforward. Suppose in our earlier example we consider the “small gap” between *section2* and *branch3* is joinable, and we specifically add a link declaration:

Example 6. Linking arbitrary branches

```
 <layout name="twoPointsLinked" start="section1">
    <start x="400" y="400" orient="30"/>
    ...
```

```
    <point id="P1" radius="small" turn="left">
        <spur>
            ...
            <point id="P2" radius="small" dir="trailing" turn="left">
                <spur>
                    <curve name="branch2" r="400" angle="155"/>
                </spur>
            </point>
            <straight name="branch3" length="500"/>
        </spur>
    </point>
    <curve name="section2" r="1500" angle="-45"/>
    ...
    <curve r="500" length="1400"/>
    <link>section2.trail branch3.trail</link>
</layout>
```

This link introduces a requirement for a "polarity shift" — a locomotive proceeding *forwards* from *section2* would find itself running in the *reverse* direction in *branch2*. To permit smooth continuous operations, our "cyber-locomotives" have a "running in the wrong-direction" property (which is *xored* with *reverse*), and when similar ports are connected with similar "down-line" properties, a dummy *swap* component is inserted in the link, which will invert this property as a vehicle transits[7]:

Figure 9. Swapping direction across links.

id	type	down	parts	length	face	trail	spur
branch1	section	false	2	1,976	P1.spur	P2.trail	
swap1	swap		0	0	section2.trail	branch3.trail	
branch2	section	true	1	1,082	P2.spur		
swap2	swap		0	0	branch3.trail	section2.trail	
branch3	section	false	1	500	P2.face	swap2.face	
section1	section	false	1	1,000		P1.face	
section2	section	false	4	4,256	P1.trail	swap1.face	
P1	points	false	1	369	section1.trail	section2.face	branch1.face
P2	points	false	1	369	branch3.face	branch1.trail	branch2.face

(*swap1* and *swap2* could in theory be the same, but the implementation is easier to use one for each direction, and the additional cost minimal.)

3.2. Computing the geometry

The original intention of the design tool was to automate the calculation of track geometry. This proved to be relatively easy, using a simple vector arithmetic package with a triple vector datatype of `x,y,orientation`[8], and the `xsl:iterate` instruction processing the track component sequences through template application as the track is "constructed". For example here is the code to process a `straight` element:

```
<xsl:template match="straight" as="map(*)" mode="makeTrack">
   <xsl:param name="start" as="map(*)"/>
   <xsl:param name="options" as="map(*)" select="map{}" tunnel="true"/>
   <xsl:variable name="length" select="@length" as="xs:double"/>
   <xsl:variable name="straight"
       select="v:new($length, 0) => v:rotateDeg($start?orient)"/>
   <xsl:variable name="end" select="v:add($start, $straight)"/>
   <xsl:variable name="path" select="p:line($start, $end)"/>
```

[7]Such an issue is faced by two-rail electric power systems on railways with such "re-entrancy"

[8]Adding a `z` (height) component would be simple, being altered by `length * gradient`. It is safe to assume that gradients will never be steep enough to make significant effects on planar (`x,y`) positions.

```
    <xsl:variable name="pieces" as="element()*">
      <g class="straight">
        <g class="schematic">
          <path d="{$path}"/>
          <xsl:sequence select="r:join($end)"/>
        </g>
        <g class="way"
           transform="translate({$start?x},{$start?y})
                      rotate({$start?orient})">
           <xsl:if test="$options?layTrack">
             <xsl:sequence select="r:straight($length)"/>
           </xsl:if>
        </g>
      </g>
    </xsl:variable>
    <xsl:sequence select="map{
         'type':string(name()),
         'orient.start' : $start?orient,
         'orient.end' : $start?orient,
         'pieces': $pieces,
         'length': $length,
         'path': $path,
         'start' : $start,
         'end': $end,
         'name': string((@name,
                         'S-'||string(accumulator-before('trackNo')))[1])
         }"
    />
</xsl:template>
```

`$start` is an input parameter which is a map whose principal members are `x`, `y` and `orient`[9]. The new end point, including its orientation, is calculated effectively by

```
v:add($start, v:new($length,0) => v:rotateDeg($start?orient))
```

where `v:rotateDeg($in,$rot)` rotates a vector (and its end orientation) by `$rot` degrees. During this operation the (SVG) graphic pieces for the schematic and the track pictures are constructed (see below) and added to the resulting map as well as other needed information, such as track section length. Each piece is named, using an `xsl:accumulator` to generate something suitable in the absence of a specific `@name` value.

This template is executed from an `xsl:iterate` instruction processing the children of a `layout` or a `spur`:

```
<xsl:template match="rail|spur|layout" as="map(*)*" mode="makeTrack">
   <xsl:param name="start" as="map(*)">
      <xsl:apply-templates select="start" mode="#current"/>
   </xsl:param>
   <xsl:iterate select="* except (start | link)">
      <xsl:param name="start" select="$start" as="map(*)"/>
      <xsl:choose>
         <xsl:when test="not(self::break)">
            <xsl:variable name="part" as="map(*)">
               <xsl:apply-templates select="." mode="#current">
                  <xsl:with-param name="start" select="$start"/>
               </xsl:apply-templates>
            </xsl:variable>
            <xsl:sequence select="$part"/>
            <xsl:next-iteration>
               <xsl:with-param name="start" select="$part?end"/>
            </xsl:next-iteration>
        </xsl:when>
```

[9]Orientation is held in degrees and converted to radians as required. SVG describes its rotations in degrees and I know fairly closely what 30°, 45° and 225° look like, but not 1.5 radians.

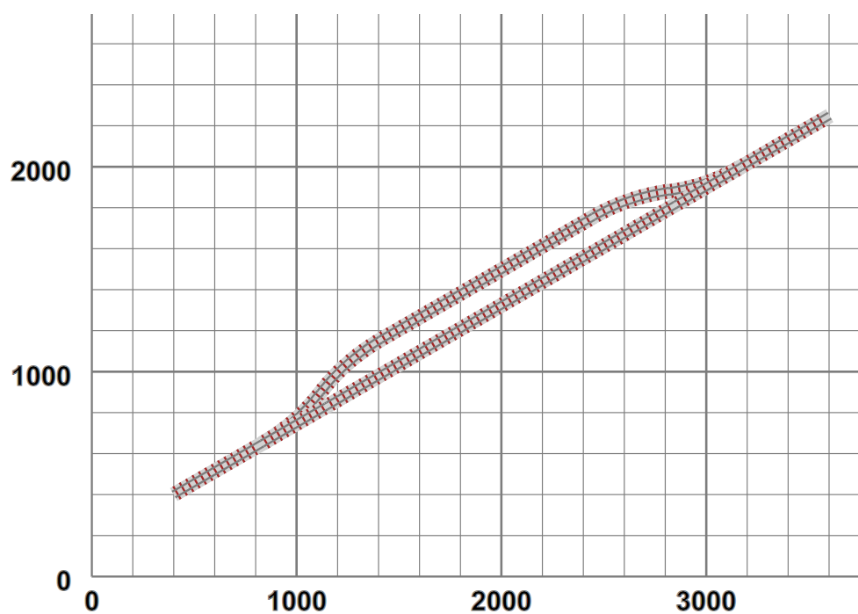
```
            <xsl:otherwise>
               <xsl:break/>
            </xsl:otherwise>
         </xsl:choose>
      </xsl:iterate>
</xsl:template>
```

For each subsequent iteration the `$start` parameter becomes the `end` property of the `$part` just generated. Needless to say processing a `curve` is similar to that for `straight`, though the calculation of the chord, end point and the appropriate SVG ellipitical arc are more complex. For the `point` we need to construct *two* sections: the *not set* (straight on) track section and its end point, and the *set* section with its attached branch line, which is constructed by a recursive call on the iteration above, with the branch `spur` element as context and the spur position and orientation as the `$start` parameter.

4. Drawing pictures

Thus far we have drawn schematic representations of the track as SVG line-based components. With a little work SVG is entirely capable of generating much more detailed views, with a lot of possibility of caching intermediate and reused sections. For example:

Figure 10. More detailed track



In this case the track is generated from a sequence of “rail-and-sleeper” subsection definitions, displayed via SVG's `use` directive:

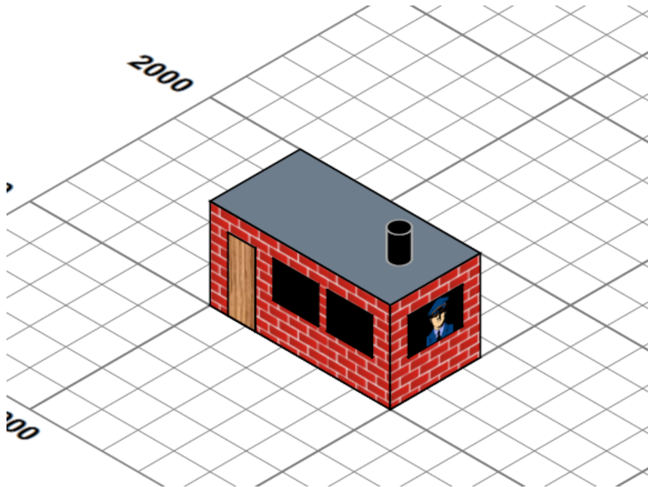
```
<g xmlns="http://www.w3.org/2000/svg" class="way"
    transform="translate(3769.549241302635,2599) rotate(30)">
    <use href="#track10" x="0" y="0"/>
    <rect class="ballast" x="360" y="-36" width="140" height="72"/>
    <use href="#sleeper" x="378" y="0"/>
    <use href="#sleeper" x="414" y="0"/>
    <use href="#sleeper" x="450" y="0"/>
    <line class="rail  SM32" x1="360" y1="-16" x2="500" y2="-16"/>
    <line class="rail  SM32" x1="360" y1="16" x2="500" y2="16"/>
</g>
```

In this case we have a “pre-built” 10-sleeper section of straight track (`#track10`), followed by “ballast”, three sleepers and two rails to display the remainder of the required length. All these are sized to the actual dimensions of the track being used. This is translated and rotated into the required start position.

4.1. Isometric Views

Planar views are useful, but they don't give a picture of what one might see, where the third dimension has some importance. Luckily an *isometric* transformation can give a view “from above and aside”. This involves applying a transform of `translate(3000,0) rotate(30) skewX(-30) scale(1,0.8660254037844387)` to the graphics and altering some pieces to support a pseudo-3D view. For example, let us add a simple building:

Figure 11. A simple building

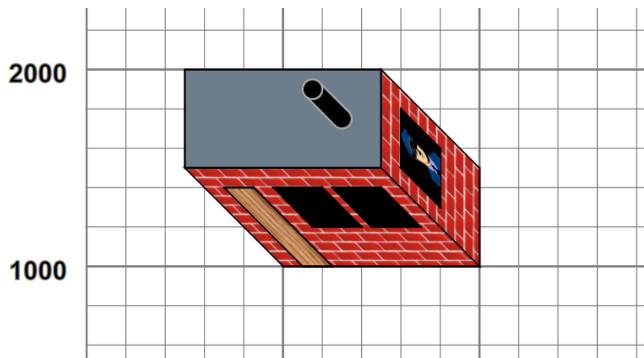


I could perhaps have looked at using a full 3D modelling package which was capable of generating SVG outputs, but my needs were modest and could perhaps be handled by a simple declarative model, processed completely with XSLT to generate suitable SVG. The building is defined by a simple XML structure of boxes and a cylinder:

```
<buildings>
  <resources> …. </resources>
  <group x="1000" y="1000"
     fill="url(#brickWall)" stroke-width="10" stroke="black">
     <box width="1000" height="500" rotateZ="0" depth="500" z="0">
        <top fill="slategrey"/>
     </box>
     <box height="1" width="150" depth="400"
        z="0" x="100" y="0" fill="url(#wood)"/>
     <box height="1" width="250" depth="200" fill="black"
        z="200" x="350" y="0" />
     <box height="1" width="250" depth="200" fill="black"
        z="200" x="650" y="0"/>
     <box width="1" height="300" depth="200" fill="black"
        z="200" x="1000" y="100">
        <east >
           <svg:image xlink:href="images/officer-in-uniform.png"
             x="100" y="0" height="200"/>
         </east>
     </box>
     <cylinder radius="50" length="150" axis="z" fill="black"
        z="500" x="800" y="250" stroke="darkgrey"/>
  </group>
</buildings>
```

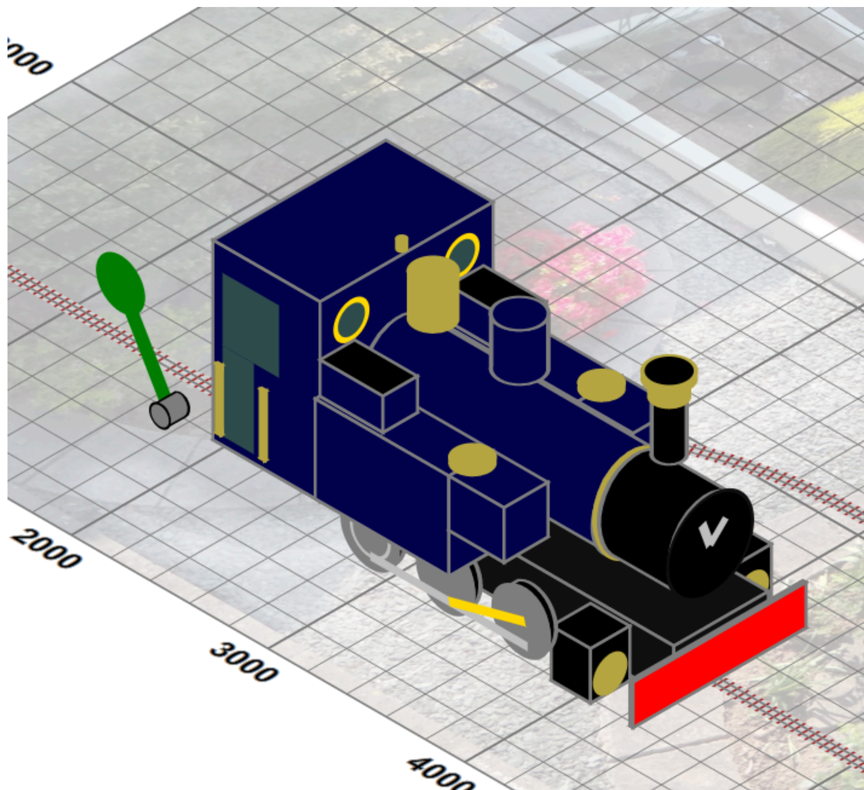
which is then used to generate an SVG group that look like:

Figure 12. An iso-orthogonal building



such that when the entire SVG group, within which lie all picture pieces (grid, plan, track etc..), is subject to isometric projection, the building appears to have depth and height. (We also produce a true orthogonal view, so we can look at the scene from "directly above".) Currently the repertoire is orthogonally-oriented rectangular blocks and cylinders, with named "faces" to which styling and content can be attached (*top*, *south* and *east* for blocks, with *bottom*, *north* and *west* normally hidden, and *surface*, *top* and *bottom* for cylinders.). Components are currently positioned absolutely and can be grouped. Using this we can build models of the complexity of:

Figure 13. The Lady Anne



which is defined by some 50 components, some of which are repeats of common substructures, implemented by bindings and interpolations of XSLT variables. This ability to style and add content to the named *faces* of the component parts is important . For example, adding the "smokebox handle" to the boiler front of *Lady Anne* merely requires:

```
<cylinder class="boilerFront" x="151" z="80" axis="x"
  radius="27" length="45">
    <end class="boilerEnd">
       <svg:g class="silver" stroke="silver" stroke-width="5">
          <line x1="0" y1="0" x2="10" y2="-10"/>
          <line x1="0" y1="0" x2="-5" y2="-14"/>
```

```
        </svg:g>
    </end>
</cylinder>
```

and the graphic components will be placed and transformed correctly to sit *in the boiler front*. As we will see later, it is critical that the SVG views of these model engines must be such that they produce the expected picture when subjected to an isometric transformation, as shown for the building, as the trajectory paths trains must follow (which are effectively *on the flat*) are themselves subjected to the same projection.

5. Interaction

The tool has two main types of interaction: animations, discussed in the next section, and view selection. Most of the view selection is based on switching the `display` style of graphical or user inteface element on and off, through controls that are generated from declarative descriptions. For example:

Figure 14. Controls for display options

```
<div name="show">
    <title>Show</title>
    <option default="">photos</option>
    <option>survey</option>
    <option>grid</option>
    <option>plan</option>
    <option>buildings</option>
 ...
 </div>
```

declares a group of controls, from which a group of labels and checkboxes are generated, some of which are preset and whose rendering is shown above. Control of display is performed by a generic XSLT template, which fields change events on the generated `input` checkboxes, all of which are class-labelled as *show*:

```
<xsl:template match="input[@class eq 'show']" mode="ixsl:onchange">
   <ixsl:set-style name="display"  object="id(@value)"
      select="if(ixsl:get(.,'checked')) then 'inline' else 'none'"/>
</xsl:template>
```

The `@value` of the `input` is taken to the be *id* of an element (either XHTML or SVG) that contains all items of the given type and *display* style modified accordingly. Generic hide/reveal controls for object with a given *class* token are supported by a similar template.

Switching between orthogonal and isometric views of the garden/plan/layout involves modifying a top-level transform attribute on the SVG and setting a class token to indicate the given view. As all (3D) components have both orthogonal and isometric views, each class-labelled, simple CSS compound rules such as `.viewISO .partORTHO,.viewORTHO .partISO {display: none;}` and `.viewISO .partISO,.viewORTHO .partORTHO {display: inline;}` ensure that only the correct class components are displayed for the currrent view.

Points obviously have state and this needs to be changed to direct trains to suitable parts of the layout. We construct an XHTML "signal box" where all the point controls are checkboxes and through which specific points can be set into *switched* or *unSwitched* classes. CSS styling ensures that the appropriate components for the given state are displayed. Sometimes determining which control effects which point can be problematic. A solution to this is to support clicking on the (SVG) points themselves, or an adjacent lever. This is achieved by the templates:

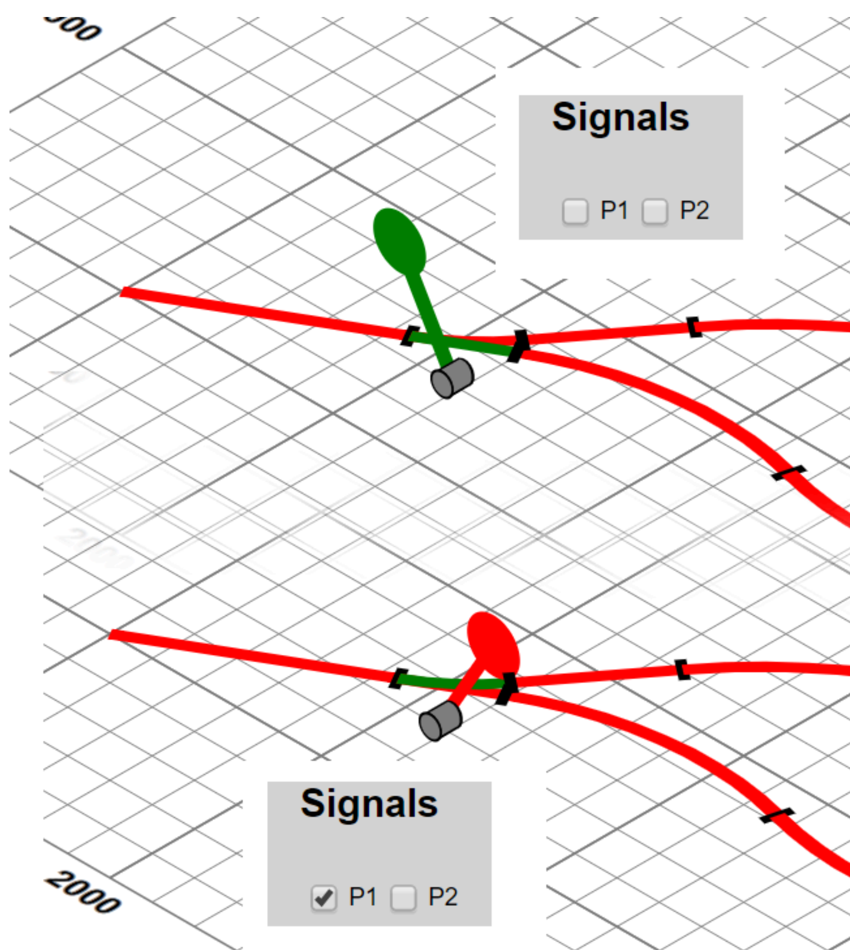
```
<xsl:template match="*[contains-token(@class, 'pointLever')]"
```

```
    mode="ixsl:onclick">
   <xsl:variable name="point"
       select="ancestor::*:g[contains-token(@class, 'point')][1]"/>
   <xsl:variable name="point.state"
       select="id($point/@id||'-state')"/>
   <xsl:sequence select="ixsl:call($point.state,'click',[])"/>
</xsl:template>

<xsl:template match="input[contains-token(@class, 'pointState')]"
    mode="ixsl:onchange">
   <xsl:variable name="checked" select="ixsl:get(., 'checked')"/>
   <xsl:sequence select="js:playAudio(id('pointChange'))"/>
   <xsl:for-each select="id('point-' || @value, .)/*:g[1]">
      <ixsl:set-attribute name="class"
         select="if($checked) then 'switched' else 'unSwitched'"/>
   </xsl:for-each>
</xsl:template>
```

where clicking on the (SVG) point lever dispatches another *click* event to the appropriate state control in the signal box. Controls in the signal box respond to changes by playing the *pointChange* sound effect and changing the (un)switched class of the actual signal, which changes which of the graphic groups is displayed:

Figure 15. Changing points with a signal box



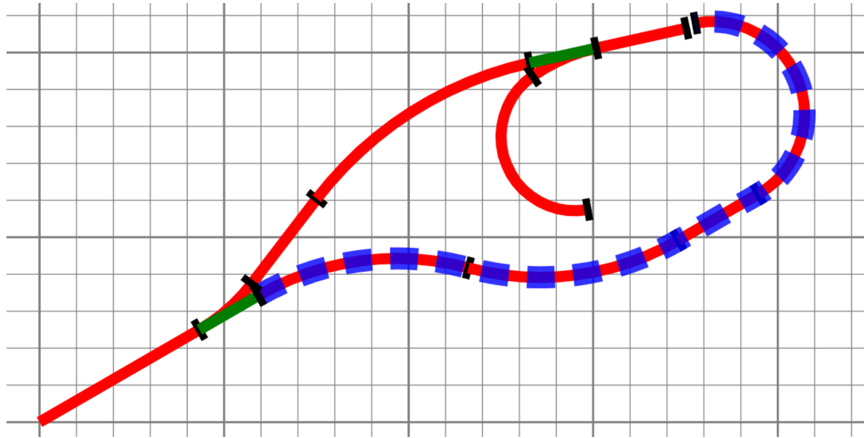
6. Animations

SVG supports animations based on SMIL event-driven models. Of particular interest in this case is the use of *path-based* animation where a given SVG group can be successively translated along a given path. As trains move along tracks, and

in our design tracks are defined by sections from which SVG `path` definitions can be constructed easily, we should be able to simulate the movement of trains around our tracks. And so it proved.

The basic animation we used is effectively "move this group *g* along this path *p* in a duration of *dur* seconds." For each section of the layout (i.e. a contiguous run of straight and curves, or the *set* and *not set* short sections of points), we calculate both a path description (the *d* property of `svg:path`) and the total length. For example the dashed blue line is the defined (single) path for the *section2* track section, for which a total length of 4,256 has been calculated :

Figure 16. A track section path



Assuming we wish our "train" to run at 100mm/s (a scale speed of ~ 7km/hr, i.e. a brisk walking pace), then the animation should take 42.5 seconds. This is achieved by forming up an `svg:animateMotion` definition element:

```
<animateMotion xmlns="http://www.w3.org/2000/svg"
   id="train.animation" xlink:href="#train"
   begin="indefinite"  fill="freeze" repeatCount="1"
   calcMode="linear" keyTimes="0;1" keyPoints="0;1"
   rotate="auto"
   dur="42.5" onend="eventEnded('train;section2.trail') >
   <mpath xlink:href="#section2.path"  />
</animateMotion>
```

The graphics group that will be subject to the animation

Conditions for the start of the animation — in this case the animation waits until it is triggered explicitly. When the animation has finished freeze the graphics state, i.e leave the graphics translated to the end of the path and do not repeat.

`keyTimes` and `keyPoints` define a piecewise-linear mapping between proportions of the duration and proportions of the total length — this is used to support moving in reverse and altering "speed".

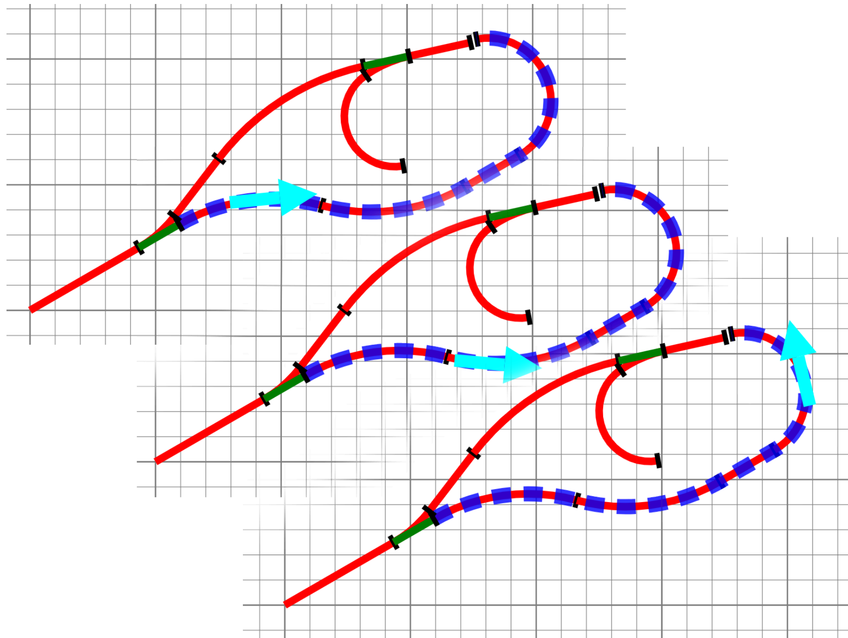
`auto` adds a rotation transform to the animated graphics corresponding to the current path tangent direction, so the graphics object "turns" along the path.

When the animation completes the global function `eventEnded()` will be executed with an argument containing information about which train has completed a move and where — in this case arriving at the *trail* port of *section2*.

A reference to the path to be followed.

The animation is started by invoking the `beginElement()` function method of the animation element through a minimal global JavaScript function. Thus our "train"(in this case a cyan arrow) progresses along *section2* as below:

Figure 17. Movement along a track section.



When the animation finishes, the `onend` statement is invoked, which is fielded by the global JavaScript function `eventEnded()`.

```
var ignoreEvent = false;
function eventEnded(e) {
   if(!ignoreEvent) {
      var event = new Event("change",{"bubbles":true});
      var store = this.document.getElementById("event");
      store.value = e;
      store.dispatchEvent(event);
  }
  ignoreEvent = false;
}
```

There are cases (described below) when we need to ignore an end event temporarily.
A (hidden) checkbox element in the DOM tree that is used to hold the event information as its `value` property.
Propogating an event that the value of the event information store has changed.

After this function has executed, the checkbox `id('event')` receives a *change* event which is caught by an XSLT template:

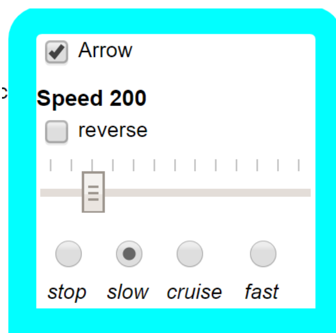
```
<xsl:template match="*:input[@id eq 'event']" mode="ixsl:onchange">
   <xsl:variable name="layout" as="map(*)"
      select="$layouts(f:radioValue('layouts', .))"/>
   <xsl:variable name="parts" select="tokenize(@value, ';')"/>
   <xsl:choose>
      <xsl:when test="exists($parts[3])">
         <!-- There is a new section to enter -->
         <xsl:call-template name="runTrain">
            <xsl:with-param name="engine" select="$parts[1]"/>
            <xsl:with-param name="trackComponentID" select="$parts[3]"/>
            <xsl:with-param name="tracks" select="$layout?tracks"/>
         </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
          <!-- There is a no new section to enter - end of the line -->
          <xsl:for-each select="id($parts[1])">
              <ixsl:set-attribute name="position" select="$parts[2]"/>
```

```
            </xsl:for-each>
            <xsl:variable name="engine" select="$parts[1]"/>
            <xsl:call-template name="stopEngine">
               <xsl:with-param name="engine" select="$engine"/>
            </xsl:call-template>
            <xsl:call-template name="reverseEngine">
                <xsl:with-param name="engine" select="$engine"/>
            </xsl:call-template>
        </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

There are a number of possible layouts, held as a named map global variable. Which is the active one is determined by the value of the *layouts* radio button set.
This template expects the value of the `event` checkbox to be a string of the form *`train;current port[;next port]`*.
If there is a next port, then the train is run on that new section from that port, on the current layout.
If not then the train is assumed to have reached the end of the line. It is stopped and the direction reversed, so that, as a convenience to the driver, "opening the throttle again" again will cause the train to move back along the section.

The trains are controlled by a simple interactive XHTML control group (obviously of class *cab*):

Figure 18. The Engine Cab




```
<div id="Arrow.cab" class="cab arrow">
  <div class="toggler">
    <input class="run" type="checkbox"
       value="Arrow" />
    <label class="text">Arrow</label>
  </div>
  <label class="title">Speed
        <span class="value">0</span></label>
  <div name="direction" class="direction">
     <div class="toggler">
        <input class="direction" type="checkbox"
           value="reverse"/>
        <label class="text">reverse</label>
      </div>
  </div>
  <input type="range" min="0" max="1200"
       value="0" list="tickmarks" />
  <div class="radio speed">
     ...
     <div class="toggler">
        <input class="speed" type="radio"
          value="200" />
        <label class="text">slow</label>
     </div>
     ...
  </div>
</div>
```

Apart from selecting a locomotive to run, the only current action is to *change its speed or direction of travel.* A number of XSLT templates detect changes in the cab input controls such as:

```
<xsl:template match="input[contains-token(@class, 'speed')]"
   mode="ixsl:onchange">
   <xsl:variable name="cab"
      select="ancestor::div[contains-token(@class, 'cab')]"/>
   <xsl:variable name="run" select="$cab//input[@class eq 'run']"/>
   <xsl:variable name="value" select="@value"/>
   <ixsl:set-property object="$cab//input[@type eq 'range']"
       name="value" select="number($value)"/>
   <xsl:for-each select="$cab//span[contains-token(@class, 'value')]">
```

```
      <xsl:result-document href="?." method="ixsl:replace-content">
         <xsl:sequence select="string($value)"/>
      </xsl:result-document>
   </xsl:for-each>
   <xsl:if test="ixsl:get($run,'checked')">
      <xsl:variable name="engine" select="$run/@value"/>
      <xsl:for-each select="id($engine)">
         <ixsl:set-attribute name="speed" select="$value"/>
      </xsl:for-each>
      <xsl:call-template name="changeVelocity">
         <xsl:with-param name="engine" select="$engine"/>
      </xsl:call-template>
   </xsl:if>
</xsl:template>
```

which detects a change in the *stop, slow, cruise, fast* radio button set. The selected speed is the `@value` of the set, which is written into a `span` element within the cab `div` and used to set the slider to a suitable point. If the engine is running (the top left checkbox checked), then the demanded speed is written as an attribute onto the selected engine object and then the *changeVelocity* template is invoked.

The key idea here is to determine *how far the current animation has progressed*, from which the remaining distance to travel can be determined. This is computed by a global JavaScript function with the animation object *a* as argument:

```
function animProgress(a) {
  if(a.getAttribute("dur")==0 ||
     a.getAttribute("dur")=="indefinite") {
    return 0;
  }
  var startTime;
  try{
    startTime = a.getStartTime();
  } catch(e) {
    return 0;
  }
  var t_ratio=(a.getCurrentTime() - startTime)/a.getSimpleDuration();
  return t_ratio;
}
```

which calculates the ratio of elapsed to total animation duration. In cases where the animation is not active (for which I can't find a simple test), the exception on finding start time is caught. Given the remaining distance and desired speed, a new duration can be determined and the animation restarted using the `keyPoints` property to start somewhere down the animation path, e.g. `keyPoints="0.5;1"` would be used for a speed change halfway along the track section[10].

The animation is restarted by invoking the `beginElement()` method — the `ignoreEvent` flag is used to prevent the implicit `endElement()` event, triggered before the restart, that would normally be used to signal completion of traversal of a section, propagating to the XSLT templates. In the case that the locomotive is running in reverse, the key points are reversed, e.g. `keyPoints="0.66;0"` would be used for a speed change one-third of the way backwards through a section.

In the absence of such speed changes a running locomotive involves animation movement along the current section until the end event is executed, fielded by the XSLT template shown earlier, which then starts animation along the next specified section. In the case of entering points, the state of the point is examined (from the status of the point control in the signal box!) and the correct path and next section determined for the animation[11]. When a locomotive enters a *swap* section, described above, its internal *running in the wrong direction* flag is inverted and it passes on to the following section.

A small number off other animation effects have been added. Firstly locomotives have wheels, which can be animated to rotate at a rate and direction suitable for their diameter and the locomotive's speed, using the animation element:

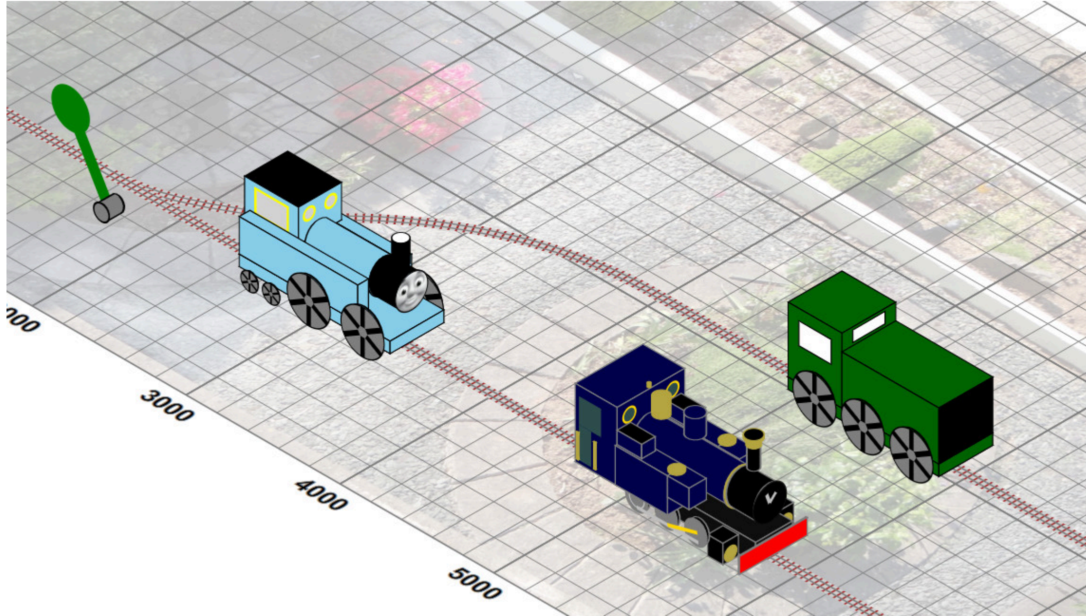
```
<animateTransform type="rotate" begin="indefinite"
   attributeName="transform" from="0" to="360"
   dur="…." attributeType="XML" repeatCount="indefinite"/>
```

[10]The current animation may itself already involve a “partial” path, as a consequence of a previous change in speed — this is determined from the existing `@keyPoints` value on the `animateMotion` element to determine the “distance to go”.

[11]Changing a point while a locomotive is moving through it will not effect the locomotive's path.

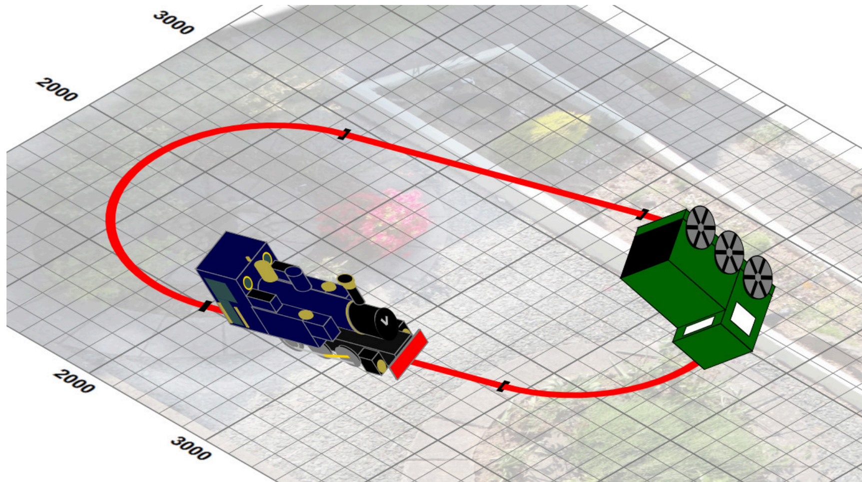
Secondly, locomotives can be given running sound effects by invoking `play()` method on an `audio` element when they start movement, and can"whistle" when they enter a (zero length) *whistle* pseudo-track section. The end point of this development was a case where multiple engines could be run on a layout, stopping, starting , reversing and changing their speed independently and altering points to move them to different sections of the layout:

Figure 19. Three engines running simultaneously



But there is a problem with the isometric view "trick" and automatic path tangent rotation:

Figure 20. On the ceiling



The animation rotation transformation is applied before the isometric projection and our 3D trick no longer works with significant rotations. How this may be overcome is discussed in the next section.

7. Developments

There are a small number of developments I have been working on, but at the time of writing they are incomplete. This section describes these ideas.

7.1. True 3D models and view rotation

The 3D model used so far is a collection of orthogonally arranged rectangular blocks and cylinders, declared in an order that reflects isometric view shadowing. For example an *engine frame* block is defined before the *boiler* cylinder, to appear

underneath it. From this model suitable SVG components can be generated to simulate a 3D view when subjected to a uniform isometric transformation. But to support a non-orthogonal rotation of such a model about the z-axis, to overcome the "on the ceiling" effect, the situation becomes somewhat more complex. There are three points to consider:

- What is a suitable graphic for a block or cylinder when rotated by θ degrees about the z-axis? A key requirement is that the "faces" model of additional styling and content must still be supported.
- As a group of 3D parts is rotated, their obscuration relationships alter and any views must accommodate this. How should a set of component parts be "depth-ordered" in the direction of the isometric view, when the ensemble is rotated significantly?
- How is the appropriate rotated view displayed as a locomotive turns?

Constructing the isometric-prepared components of a rotated block is a little tricky. The top surface is always visible and can just be rotated as required. Ignoring any visibility of the base, only two of the four vertical sides will be visible dependent upon rotation change ranges of 45° and 135°. Each visible face is subjected to additional scaling and skew dependent on the rotation angle, so that it is correctly sized, positioned and any additional content "stays in place". The situation for horizontally aligned cylinders is very much more complex, and at the time of writing is work in progress.

To "view-order" an ensemble of rotated components it would be helpful if a (possibly multiple) value can be computed that can be used as sort keys to arrange the parts into appropriate order using `xsl:perform-sort/xsl:sort+`. This can be so for some very simple cases, but in general parts must be pairwise-compared, which requires some sorting function that uses a *compare* function, rather than a key-generator. Sadly, XPath sort functions all use a "key" model, so a generic XSLT higher-order pairwise sorting function may have to be constructed.

Calculating the rotation views *on the fly* would be catastrophically expensive, so the solution chosen is to generate a series of groups, each corresponding to a defined angle of rotation and labelled suitably (e.g. `class="rotate-45"` for a view rotated by -45°). It would also be possible to generate the set of views offline and include in the runtime. However they can be sizeable — an interval of 5°, which certainly doesn't appear "smooth" would require 72 separate versions.

Assuming there is such a series of views of an engine, we need to arrange for the display property of the (approximately) correct rotation view to be switched from *none* to *inline*. But we do know for a given locomotive which section it is in and can map from the proportion of the animation completed to the tangential orientation at that point. (As we use only straights and circular arcs, the tangent angle is a piecewise linear function of the "section proportion", running from 0 to 1. This profile is added to the map entry for the section.) Given that the speed of the engine is known, we can thus predict how long it will be until the current rotation view should be superseded by the next one. This is enabled through a template `rotateTrain` which both makes visible the suitable view and schedules a further `rotateTrain` call after a suitable wait.

7.2. Collision detection, *a.k.a.* train crashes

As designed, my locomotives are ætheral beings, able to glide seamlessly and smoothly through each other. To prevent this, we need to detect collision or interference. SVG does have some primitive collision detection based on bounding box overlap, but given the isometric 3D nature of our engines, this is unlikley to be accurate, and certainly over-enthusiastic. Moreover, normal movement of our engines is both highly restrained, i.e. to track sections, and predictable, as they travel at known rates.

A simple approach, ignoring engine "size" and treating them as point entities, is to consider only cases where two (or more) engines are in the same section[12], travelling in either the same or opposite directions. Such a case can be checked when either a locomotive enters a new section, or the speed of an engine is changed. In such circumstances, we know both where, in distance, each locomotive is, and how fast they are approaching each other. Hence in the case of a predicted collision we can schedule an action (using `ixsl:scheduleAction`) to trigger a "crash notification" after the required interval. However there is also the difficulty of a subsequent speed change altering this — this requires the ability to delete some of the currently active scheduled actions, which has proved highly problematic.

7.3. Difficulties

Apart from the headache-inducing issue of calculating the geometry of the edege of the visible curved surface of a rotated cyclinder, most of the difficulty has been managing the animations and events. In particular it appears that an active animation cannot be stopped and deleted or restarted without invoking any associated `onend` event. The temporary solution, of dubious robustness, uses a global flag to suppress subsequent event propagation.

[12]Much of nineteenth-century railway signalling development was of course to stop such a situation happening in the first place.

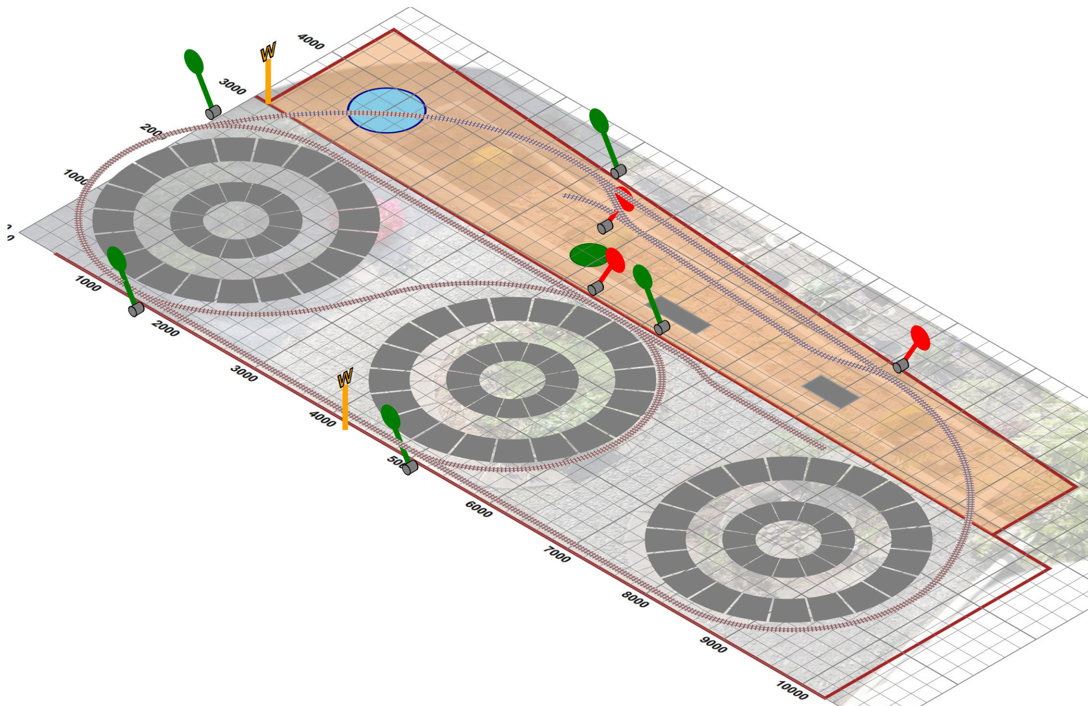
8. Conclusion

I originally built a small tool, using XSLT, that "did the geometry calculations" for a layout I was designing. A graphical view is always helpful, and generating SVG to do so was straighforward. Developing the isometric view led towards a more pictorial aspect to the output. Adding very simple animation opened the possibility of building something more akin to a "train set", and showed some of the ways controls and active state could be mixed in an XSLT/Saxon-JS/SVG/browser environment. And this led to the idea of a demonstration at MarkupUK 2018...

The implementation needed a very small number of global JavaScript functions, that were invoked in XSLT/XPath expressions through the Saxon-JS function mapping namespace `http://saxonica.com/ns/globalJS`. All the rest of the code is XSLT3.0, with Saxon-JS extensions, generating all necessary XHTML and SVG structures, with templates fielding and processing events both from interaction and animation. Once up and running, the system is of course stateful — the speed, direction and current track section of engines, the switched *set* state of points etc. This state information is stored as attributes on the DOM tree.

Did it help with the original purpose — designing a garden railway? Well this was the layout design demonstrated at Markup2018:

Figure 21. The layout as proposed



and this is what currently exists:

Figure 22. Lady Anne on the Garden Line

Without Saxon-JS this project wouldn't have even been attempted and thanks are due to my colleagues Mike Kay and Debbie Lockett for the excellence of that product. The author is of course extremely grateful for the many votes cast in his direction at last year's MarkupUK DemoJam — without them he wouldn't have had to write this paper.

References

[1] Debbie Lockett and Michael Kay. *Saxon-JS: XSLT 3.0 in the Browser.*. Balisage: The Markup Conference . 2016. https://doi.org/10.4242/BalisageVol17.Lockett01.

[2] *Scalable Vector Graphics (SVG) 1.1 (Second Edition).* 2011. World Wide Web Consortium (W3C). https://www.w3.org/TR/SVG11/.

[3] *XSL Transformations (XSLT) Version 3.0*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xslt-30/.

Taking Schematron QuickFix To The Next Level

Octavian Nadolu, Oxygen XML Editor

Abstract

The Schematron QuickFix (SQF) [http://schematron-quickfix.github.io/sqf/spec/SQFSpec.html] language can be used to improve efficiency and quality when editing XML documents. You can define actions that will add complex XML structure in your documents, make modifications in multiple places, or actions that will convert an XML structure into another. These changes are made by keeping the document structure valid and conforming to your project specification, and will help the content writer add content more easily and without making mistakes.

To build complex actions, you can mix the Schematron QuickFix language with ISO Schematron [http://schematron.com/], or with different versions of XSLT [https://www.w3.org/TR/xslt/all/] and XPath [https://www.w3.org/TR/xpath/all/], or you can define your own extensions using programming languages such as Java. This means that you can perform modifications in multiple external documents, or display dialog boxes to get input from your users, or use complex algorithms for processing the content of the documents.

The Schematron QuickFix is a modular language. Even though it is simple (it only has four types of operations that can be performed: add, delete, replace, and string replace), new types of operations can be created (such as wrap, unwrap, rename, or join). This can be implemented by using abstract quick fixes and by creating libraries of quick fixes that can be reused for various different XML vocabularies (such as DITA, DocBook. TEI). By using a library of quick fixes, it will help the Schematron QuickFix developers to create the actions more easily and reuse the quick fixes created by others.

This presentation will focus on some interesting use-cases and Schematron QuickFix examples that can be easily adapted to your projects. The examples will include both abstract quick fixes and complex quick fixes that use XSLT and Java.

1. Introduction

Schematron is a powerful language, and one of the reasons is because it allows the schema developers to define there own custom messages. The messages can also include hints to help the user correct the problem, but this operation must be performed manually. Correcting the problem manually is inefficient and can result in additional problems.

The SQF language allows you to define actions that will automatically correct the problem reported by Schematron assertions. This will save you time and money, and will help to avoid the potential for generating other problems.

2. Schematron QuickFix Language

Schematron QuickFix (SQF) is a simple language that allows the Schematron developer to define actions that will correct the problems reported by Schematron rules. SQF was created as an extension of the Schematron language. It was developed within the W3C "Quick-Fix Support for XML Community Group". The first draft of the Schematron Quick Fix specification was published in April 2015, second draft in March 2018, and it is now available on the W3C Quick-Fix Support for XML community group [https://www.w3.org/community/quickfix/] page.

The actions defined in a Schematron QuickFix are called operations. There are four types of operations defined in the SQF language that can be performed on an XML document: *add*, *delete*, *replace*, and *string replace*. The operations must perform precise changes in the documents without affecting other parts of the XML document.

Example 1. A Schematron Quick Fix that adds a 'bone' element as child of the 'dog' element

```
<sch:rule context="dog">
    <sch:assert test="bone" sqf:fix="addBone">
        A dog should have a bone.</sch:assert>
    <sqf:fix id="addBone">
        <sqf:description>
            <sqf:title>Add a bone</sqf:title>
        </sqf:description>
        <sqf:add node-type="element" target="bone"/>
    </sqf:fix>
</sch:rule>
```

3. Use Cases

3.1. Type of Users

It is important to have quality control over the XML documents in your project. You can do this using a Schematron schema in combination with other schemas (such as XSD, RNG, or DTD). Schematron solves the limitation that other types of schema have when validating XML documents because it allows the schema author to define the tests and control the messages that are presented to the user. It makes validation problems more accessible to users and it ensures that they understand the problem.

However, correcting the validation problems in the XML documents can sometimes be difficult for a content writer. Many content writers are not XML experts, but they know the context of the document very well and they are experts in their domain. If a problem occurs when they write content in the document, they need to fix it correctly without adding new ones.

On the other hand, an XML expert knows the syntax very well and knows how to fix the problem, but they may not be familiar with the domain of the content writer. In this case, since the XML expert and content writer will need to work together to correct the problems, this will introduce extra costs and take more time.

Using SQF, you can provide in-place actions to help the content writer correct the problems by themselves without involving the XML expert, and without producing new errors. This will solve the problem faster and companies will spend less money. The XML expert can focus on developing new rules and quick fixes for the content writer.

3.2. Generate Content Using XPath

The content inserted by a Schematron quick fix operation can be static, such as text or tags specified in the quick fix operation, or can be generated dynamically depending on the current node content or depending on the other nodes from the document. To generate dynamic content, you can use XPath expressions in the value of the *@select*" attribute for the *sqf:add*, *sqf:replace*, or *sqf:stringReplace* operations.

By using XPath expressions, you can perform complex operations and generate content depending on the existence of some elements in the document or use content from other documents. You can benefit from more than 200 XPath built-in functions. There are functions for string values, numeric values, date and time comparison, node manipulation, and much more.

For example, perhaps you have a rule that checks if a section *element* has an *@id* attribute specified and reports any occurrences as a problem. Then, you can have a quick fix that adds an *@id* attribute on the current *section* element. The value of the attribute can be created by using the title section and removing any special characters or spaces from the title. In case the section does not have a title, you can generate a random value for the id.

Example 2. Use the title value as ID or generate a random ID

```
<sqf:fix id="addID">
  <sqf:description>
      <sqf:title>Add ID to the current section</sqf:title>
  </sqf:description>
  <sqf:add node-type="attribute" target="id"
      select="if (exists(title) and string-length(title) > 0)
      then substring(lower-case(replace(replace(normalize-space(string(title)),
                                                '\s', '_'),
      '[^a-zA-Z0-9_]', '')), 0, 50)
      else generate-id()"/>
</sqf:fix>
```

In case you want to add an *@id* attribute on all the *section* elements from the document, you can create a quick fix that matches all the selection elements that do not have an id attribute. To do this, you need to create a similar quick fix as the previous one and just add a *match* attribute on the *sqf:add* operation with the value: *//section[not(@id)]*.

Example 3. Match all the section elements from the document and add an ID

```
<sqf:fix id="addID">
  <sqf:description>
      <sqf:title>Add ID to the all entries from document</sqf:title>
  </sqf:description>
  <sqf:add match="//section[not(@id)]" node-type="attribute" target="id"
      select="if (exists(title) and string-length(title) > 0)
      then substring(lower-case(replace(replace(normalize-space(string(title)),
                                                '\s', '_'),
      '[^a-zA-Z0-9_]', '')), 0, 50)
      else generate-id()"/>
</sqf:fix>
```

You can also use XPath functions such as *document-uri(/)* in case you want to add an *id* attribute with the same value as the file name.

Example 4. Get the current document file name and use it as ID

```
<sqf:fix id="addFileID">
  <sch:let
    name="reqId"
    value="substring-before(tokenize(document-uri(/), '/')[last()], '.')"/>
  <sqf:description>
      <sqf:title>Set "<sch:value-of select="$reqId"/>" as ID</sqf:title>
  </sqf:description>
  <sqf:add node-type="attribute" target="id" select="$reqId"/>
</sqf:fix>
```

3.3. Change Text Using Regular Expressions

To make changes in text content, you need to use regular expressions. The *sqf:stringReplace* operation allows you to specify a regular expression to find sub-strings of text content and replace them with nodes or other strings. The regular expression need to be specified in the value of the *regex* attribute.

For example, suppose you have a rule that checks if a link is added as text content and not marked as a link. You can create a quick fix replace the text link with an *xref* element that has the value of the text link. This will allow you to automatically fix the links from the text and mark them as clickable links.

Example 5. Create a clickable link from a text link

```
<sqf:fix id="convertToLink">
  <sqf:description>
      <sqf:title>Convert '<sch:value-of select="$linkValue"/>' text link
                 to xref</sqf:title>
  </sqf:description>
  <sqf:stringReplace regex="(http|www)\S+">
      <xref href="{$linkValue}" format="html"/>
  </sqf:stringReplace>
</sqf:fix>
```

To get the value of the link from the text content, you can use the *xsl:analyze-string* instructions. This will allow you to process the string content and obtain the content that you need. Then you can make the change in the document using the *stringReplace* operation.

Example 6. Obtain the value of the link from text

```
<xsl:variable name="linkValue">
  <xsl:analyze-string select="." regex="(http|www)\S+">
    <xsl:matching-substring>
      <xsl:value-of select="regex-group(0)"/>
    </xsl:matching-substring>
  </xsl:analyze-string>
</xsl:variable>
```

Therefore, using regular expressions and the *sqf:stringReplace* operation, you can tag content or you can replace text content with other text content.

3.4. Using XSLT to Generate Content

A Schematron QuickFix can also be used to improve efficiency when adding content in XML documents. You can define actions that will automatically add a new XML structure in the document at a valid location, or actions that will convert an XML structure into another. These types of actions will help the content writer add content more easily and without making mistakes.

Using XSLT, you can create complex actions. For example, actions that will correct a table layout and use complex XSLT processing, or actions that use data from other documents or from a database. This will allow the content writer to focus on the content of the document while the quick fixes will help them to easily insert XML structure or to fix various issues that can appear during editing.

If you have a rule that checks if a table layout is correct and you have the same number of cells on each row, you can create a quick fix that adds the missing cells. To do this, you can use XSLT to calculate and generate the exact number of cells that are missing on each row.

Example 7. Add the missing cells from a table

```
<sqf:fix id="addCells">
  <sqf:description>
      <sqf:title>Add enough empty cells on each row</sqf:title>
  </sqf:description>
  <sch:let name="reqColumsNo" value="max(.//row/count(entry))"/>
  <sqf:add match="row" position="last-child">
      <sch:let name="columnNo" value="count(entry)"/>
      <xsl:for-each select="1 to xs:integer($reqColumsNo - $columnNo)">
          <entry/>
      </xsl:for-each>
  </sqf:add>
</sqf:fix>
```

3.5. Ignore Schematron Checks

Schematron allows you to define your own custom validation rules for XML documents. You can even set different severities for the reported problems, such as *fatal*, *error*, *warn*, *info*. But in some situations, the users want to ignore some of this rules. For example, if the problem severity is *warn*, maybe it is not something important, or if the problem severity is *info*, maybe is more like a guideline for the users.

There is no built-in implementation to ignore reported Schematron problems. But this mechanism can be implemented in different ways using the current Schematron support and quick fixes. To do this, you need to uniquely determine the rule that you want to ignore and you can do this by assigning an ID to the rule check. Then you need to store this ID in the file or in a separate file or option to avoid triggering the Schematron check in the future. You can assign the ID to a *rule* element. This means that you will be able to ignore all the asserts from that rule. Alternatively, you can assign an ID to each *assert/report* element, and this means that you will be able to ignore each assert/report from the rule separately.

In my implementation, I decided to assign an ID for each assert/report check. The ID is specified in the value of the *ruleCheckId* variable, and it will be used by the *isRuleIgnored()* function to check if the current rule is marked as ignored. A better way to implement this is to use the *id* attribute value specified on the *assert/report* element, but this means that it will not work with the current Schematron implementation that does not process this ID value. In the following example, the report is triggered only if the *boldCheck* rule is not marked as ignored.

Example 8. Schematron rule that is triggered only if is not ignored

```
<sch:let name="ruleCheckId" value="'boldCheck'"/>
<sch:report test="not(func:isRuleIgnored(., $ruleCheckId)) and exists(b)"
  sqf:fix="resolveBold ignoreRule ignoreRuleGlobal" role="warn"> Bold
  element is not allowed in title. </sch:report>
```

To store the ID of the rule check that is ignored in the XML document, I used a processing instruction with the name *SuppressRule*. The processing instruction is added before the current node, in case you want to ignore only the current rule check, or at the end of the document in case you want to ignore the current rule in the entire document. Another solution would be to store this ID in a separate document to avoid adding processing instructions in the edited XML document.

The following XSLT function checks if there is a processing instruction with the name *SuppressRule* before the current element or at the end of the document, and if the ID of the current rule is specified in the processing instruction. It returns *true* if the rule is specified as ignored.

Example 9. XSLT function that verifies if an assert/report is marked as ignored

```
<xsl:function name="func:isRuleIgnored" as="xs:boolean">
  <xsl:param name="node"/>
  <xsl:param name="ruleId"/>
  <xsl:value-of
    select="($ruleId =
        $node/preceding-sibling::processing-instruction()[name() =
            'SuppressRule']/tokenize(., ' ')
      or $ruleId =
        /processing-instruction()[name() =
            'SuppressRule']/tokenize(., ' '))"/>
</xsl:function>
```

To mark a rule as ignored, you can use quick fix actions. You can define a quick fix that will marked as ignore the current rule check by adding a processing instruction before the current element, with the name *SuppressRule* and the value of the rule ID. In case the processing instruction is already added for other ignored rules, it will concatenate the current rule ID to the existing ones.

Example 10. Quick fix that adds the current check to the ignore list

```
<sqf:fix id="ignoreRule" role="delete">
  <sqf:description>
    <sqf:title>Ignore current rule</sqf:title>
  </sqf:description>
  <sch:let
    name="ignoredRulePI"
```

```
    value="preceding-sibling::processing-instruction()[name() = 'SuppressRule']"/>
  <sqf:delete match="$ignoredRulePI" use-when="$ignoredRulePI"/>
  <sqf:add position="before">
    <xsl:processing-instruction
     name="SuppressRule"
     select="concat($ignoredRulePI, ' ', $ruleCheckId)"/>
  </sqf:add>
</sqf:fix>
```

You can also define a similar quick fix action that will ignore the current rule check for the entire document by adding the processing instruction at the end of the document. You can also define a quick fix action that will add the rule ID in a separate file and specify if the rule is ignored for the current file or for the entire project.

3.6. SQF User Input Dialog

Sometimes, when executing a quick fix action, you need to interact with the user and get input from them, or make them choose between different options. The quick fix specification defines the *sqf:user-entry* for such interactions. The `sqf:user-entry` element defines a value that must be set manually by the user.

If you need multiple values to be specified by the user, you can define multiple `user-entry` elements. For each *user-entry*, a dialog box will be displayed where the user can specify values. The `user-entry` element can be used as an XPath variable where the XPath variable is the name of the `user-entry`.

In the following example, the quick fix action defines two user entries, one that will allow the user to specify the value of the hypertext link, and one the will allow them to specify the link title. When the quick fix will be executed, two dialog boxes will be displayed, one for the link value and a second one for the link title. The order of the dialog boxes is define by the order of the defined user entries in the quick fix.

Example 11. Quick fix action that presents two user entry dialogs

```
<sqf:fix id="addAnchorAndTitle">
  <sqf:description>
    <sqf:title>Surround image with a hypertext link</sqf:title>
  </sqf:description>
  <sqf:user-entry name="href">
    <sqf:description>
      <sqf:title>Enter anchor href value:</sqf:title>
    </sqf:description>
  </sqf:user-entry>
  <sqf:user-entry name="linkTitle">
    <sqf:description>
      <sqf:title>Enter link title:</sqf:title>
    </sqf:description>
  </sqf:user-entry>
  <sqf:replace>
    <a href="{$href}" title="{$linkTitle}">
      <xsl:copy-of select="."/>
    </a>
  </sqf:replace>
</sqf:fix>
```

In a quick fix, you can also use you own extensions using programming languages such as Java. This will allow you to present your own custom dialog boxes to get input from the user or make complex processing that is not possible with the current Schematron or XSLT support.

For example, you can invoke the *JFileChooser* dialog box to allow the user to choose a file from the file system, and then use the file path in the document. To do this, you need to create a quick fix action that will call an XSL template that will present the dialog box and obtain the selected value from the user.

Example 12. Quick fix that displays a browse dialog

```
<sqf:fix id="addSrc">
  <sqf:description>
```

```
    <sqf:title>Browse and add @src attribute</sqf:title>
  </sqf:description>
  <sqf:add node-type="attribute" target="src">
    <xsl:call-template name="browse"/>
  </sqf:add>
</sqf:fix>
```

To display the dialog box from XSLT, you need to create a new instance using a *new()* call, and then use the *showOpenDialog()* to show a dialog box that will allow the user to choose a file. Depending on whether the user decides to cancel the dialog box or choose a file, you can obtain the current selected file using the *getSelectedFile()* call. The result of the template will be sent to the quick fix action that will inset it in the document.

Example 13. Quick fix that displays a browse dialog

```
<xsl:template name="browse">
  <xsl:variable name="dialog" select="jBrowse:new()"/>
  <xsl:variable
    name="result"
    select="jBrowse:showOpenDialog($dialog, $dialog)"/>
  <xsl:if test="$result = 0">
    <xsl:value-of
     select="jBrowse:getSelectedFile($dialog)"/>
  </xsl:if>
</xsl:template>
```

4. Abstract Quick Fixes

The Schematron QuickFix language is a simple language, and has just four types of operations that can be performed (add, delete, replace, and string replace). Being a simple language, it is easy to learn and use, and also easy to implement by applications.

Sometimes the developers that create the quick fixes need to use other types of operations (such as wrap, unwrap, rename, or join). They expect to have these operations defined in the language. Defining more operations in the language will help them create the quick fixes more easy, but this means that the language will be more complicated to learn and harder to be implemented by applications. A solution to this problem is to define a library of generic quick fixes that can be used for other types of operations.

A library of quick fixes can be implemented using abstract quick fixes. An abstract quick fix can be defined as a quick fix that has abstract parameters defined at the beginning of an `sqf:fix` element.

Example 14. Schematron abstract quick fix that can be used to rename a generic element

```
<sqf:fix id="renameElement" role="replace">
    <sqf:param name="element" abstract="true"/>
    <sqf:param name="newName" abstract="true"/>
    <sqf:description>
        <sqf:title>Rename '$element' element in '$newName'</sqf:title>
    </sqf:description>
    <sqf:replace
        match="."
        target="$newName"
        node-type="element"
            select="node()"/>
</sqf:fix>
```

An abstract quick fix can be instantiated from an abstract pattern. The pattern must have all the parameters declared in the quick fix, and the quick fix must declare all the abstract parameters that are used. Abstract parameters cannot be used as normal XPath variables. The reference of the abstract parameter will be replaced by the value specified in the abstract pattern.

Example 15. Schematron abstract pattern that reference an abstract quick fix

```
<sch:pattern id="elementNotAllowed" abstract="true">
```

```
    <sch:rule context="$element">
        <sch:assert test="false()" sqf:fix="renameElement">
            Element '$element' not allowed, use '$newName' instead.
        </sch:assert>
    </sch:rule>
</sch:pattern>
```

The abstract pattern can be instantiated by providing different values for the parameters. Therefore, you can quickly adapt to multiple variants of XML formats and provide rules and quick fixes that will allow the user to correct the problems.

Example 16. Schematron abstract pattern instantiation

```
<sch:pattern is-a="elementNotAllowed">
    <sch:param name="element" value="orderedlist"/>
    <sch:param name="newName" value="itemizedlist"/>
</sch:pattern>
```

Another solution for providing other quick fix actions without using abstract patterns is to use the `sqf:call-fix` [http://schematron-quickfix.github.io/sqf/publishing-snapshots/March2018Draft/spec/SQFSpec.html#param.call-fix] element.

5. Multilingual Support in SQF

The second draft of the Schematron QuickFix specification comes with an important addition, the localization concept for quick fixes. It is based on the Schematron localization concept, but it is more flexible.

A new attribute was added for the `sqf:title` and `sqf:p` elements, the `@ref` attribute. In the value of the `@ref` attribute, you can specify one or more IDs that point to different translations of the current phrase. The specification does not restrict the implementations of the `@ref` attribute to a specific reference structure.

Example 17. Schematron QuickFix that has multilingual support

```
<sqf:fix id="addBone">
    <sqf:description>
        <sqf:title ref="fix_en fix_de">Add a bone</sqf:title>
        <sqf:p ref="fix_d_en fix_d_de">Add a bone as child element</sqf:p>
    </sqf:description>
    <sqf:add node-type="element" target="bone"/>
</sqf:fix>
```

One possible implementation of the multilingual support in SQF is to use the Schematron `diagnostics` element. You can define a diagnostic for each referenced `id` and specify the language of the diagnostic message using the `xml:lang` attribute on the `sch:diagnostic` element or on its parent.

Example 18. Schematron diagnostics

```
<sch:diagnostics>
    <sch:diagnostic
       id="fix_en"
       xml:lang="en">Add a bone</sch:diagnostic>
    <sch:diagnostic
       id="fix_de"
       xml:lang="de">Fügen Sie einen Knochen hinzu</sch:diagnostic>
</sch:diagnostics>
```

This implementation conforms with the Schematron standard that also uses diagnostic for localization. It is easier to translate the messages because the Schematron messages and quick fix messages are kept together, and another important aspect is that it will be the same implementation used for both Schematron and SQF messages.

There are also some issues with this implementation. One of them is that you cannot have IDs with the same name in your document because the diagnostic uses XML IDs. Another issue is that SQF will depend on the Schematron language and cannot be encapsulated separately.

Another implementation of quick fix localization is to use Java Property Files. In this case, the localized text phrases should be stored in external files, grouped by language. These files should be placed parallel to the Schematron schema with the name pattern `${fileName}_${lang}.xml`. The `${fileName}` should be the name of the Schematron schema but without the extension. The `@ref` attribute from the quick fix must reference the property key.

Example 19. Java Property File for German translation

```
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <entry key="dog.addBone.title">Füge einen Knochen hinzu</entry>
    <entry key="dog.addBone.p">Der Hund wird einen Knochen erhalten.</entry>
</properties>
```

In contrast to the references to `sch:diagnostic` elements, in this case, it is not necessary to make any changes in the Schematron schema to introduce a new language. You just have to add a file with the name (for example `localized_fr.xml`) in the same folder of the Schematron file (for example `localized.sch`).

However, this needs a different implementation than the standard Schematron, and the Schematron messages and the SQF messages will be in different locations.

6. Generate Quick Fixes Dynamically

Another important addition in the second draft of the Schematron QuickFix specification is the possibility to define generic quick fixes. Using a generic quick fix, the developer can generate multiple similar fixes using the values provided by an XPath expression.

In the first draft of the SQF specification, it was not possible to have a dynamic amount of quick fixes for a Schematron error. The developer should specify the quick fixes presented for a Schematron error. They could only control whether or not a quick fix will be presented to the user by using the `@use-when` attribute.

To create a generic quick fix, the SQF developer needs to set the `use-for-each` attribute for the `sqf:fix` element. The value of the `use-for-each` attribute must be an XPath expression. For each item of the evaluated XPath expression, a quick fix is provided to the user. The context inside of the quick fix will be the current Schematron error context. To access the value of the current item from the XPath expression, a built-in variable `$sqf:current` can be used.

Example 20. A Generic QuickFix that provides a quick fix to remove each item from a list

```
<sqf:fix id="removeAnyItem" use-for-each="1 to count(li)">
    <sqf:description>
        <sqf:title>Remove item #<sch:value-of select="$sqf:current"/></sqf:title>
    </sqf:description>
    <sqf:delete match="li[$sqf:current]"/>
</sqf:fix>
```

Using generic quick fixes, the SQF developer can now provide a dynamic number of fix actions depending on a set of values from the current document or they can use an XPath expression to get the values from external documents.

7. Conclusion

Schematron has become a very popular language in the XML world. In the last few years, Schematron started to be used more and more and in numerous domains. One of the reasons Schematron started to become more popular is because you now also have the ability to define actions that will allow the user to correct the problem, rather than just presenting an error message. SQF has made Schematron more powerful and became a powerful language in itself.

SQF has also evolved by adding the multilingual support or the support to generate quick fixes dynamically. The specification has became more accurate and the implementations more stable. Developers are discovering new ideas for working with Schematron quick fixes even outside of Schematron language.

Bibliography

[SQF Spec] Schematron Quick Fix specification, http://schematron-quickfix.github.io/sqf

[SQF Site] : Schematron Quick Fix official site http://www.schematron-quickfix.com/

[SCH] Schematron official site http://schematron.com/

[SCH Spec] Schematron specification https://standards.iso.org/ittf/PubliclyAvailableStandards

Accessibility Matters

Tony Graham, Antenna House, Inc.

Abstract

XML, by itself, does not have any support for accessibility. XML is extremely flexible, but it needs to flex in the right directions if it is going to support the information necessary to make a document accessible. This is a guided tour of some of the features of the HTML, Web Content Accessibility Guidelines (WCAG), and PDF/UA (Universal Accessibility) standards. It concentrates on file formats rather than User Agent behaviour, since the information needed to make accessible HTML or PDF usually needs to be included in, or able to be inferred from, the source XML.

However, it's rarely the raw XML that is presented to users. This paper also strays into some aspects of styling the content to make it more accessible.

1. What is accessibility?

Accessibility is the inclusive practice of making content more accessible to people with disabilities. Accessibility "involves a wide range of disabilities, including visual, auditory, physical, speech, cognitive, language, learning, and neurological disabilities. Accessible content is also more usable by older individuals with changing abilities due to aging and will often improve usability for users in general." [WCAG2.1]

2. Standards for accessibility

Relevant standards include:

- Matterhorn Protocol [Matterhorn]
- PDF File 508 Checklist (WCAG 2.0 Refresh) `https://www.hhs.gov/web/section-508/making-files-accessible/checklist/pdf`
- PDF Techniques for WCAG 2.0 `https://www.w3.org/TR/2014/NOTE-WCAG20-TECHS-20140408/pdf.html`
- Web Content Accessibility Guidelines (WCAG) 2.1 [WCAG2.1]

3. Accessibility in, accessibility out

The well-known acronym GIGO for "Garbage in, garbage out" [GIGO] refers to poor-quality input producing poor-quality output. The corollary for accessibility is that good accessibility in your output requires that the information necessary for that good accessibility needs to be present in your data. Chandi Perera of Typefi likens adding accessibility at the end to baking a cake and then trying to take the nuts out of the cake when serving because someone in the room has a nut allergy [Perera]. Trish Ang of Slack put it more simply: "it's easier to add the blueberries in before you've baked the muffin." [Ang]

Software, even the software doing the final formatting, can help by adding some of the accessibility information. In a JATS-List post, Bruce Rosenblum of Inera stated that his conversion software has "been setting scope attributes in JATS XML files for years for customers who need section 508 compliance and it's met their requirements." [Rosenblum] However, relying on software at the end of the process to fulfil the letter, but not the spirit, of accessibility requirements can, to continue the food analogies, leave everyone with a bad taste in their mouth.

One of the best known accessibility requirements for web pages is to provide alternate text for images and links. Formatting software such AH Formatter can include alternate text when generating Tagged PDF output, but a formatter can only work with what's in its XSL-FO or HTML input. The AH Formatter Online Manual notes "It is the FO/HTML creator's responsibility to provide meaningful alternate text." [TaggedPDF] Since the formatter can't stop and ask for the input to be edited and then resume, so it has to take increasingly desperate steps to find alternate text to use for an element:

- If the `axf:alttext` property is present and not empty, its value is used.
- Otherwise, if the element contains text, that text is used.
- Otherwise, if the source is HTML and the element has a `title` property, its value is used.
- Otherwise, the value of the `role` property, if present and not empty, is used.
- Otherwise, a single space character (U+0020) is used.

The `role` property is a poor alternative, since it is not designed for use as alternate text, and, obviously, a single space character is even less meaningful, but if the source document does contain meaningful accessibility information, then that accessibility information can be included in the output.

4. HTML

Source XML in formats such as DocBook, JATS, or DITA are often transformed into HTML for delivery, inclusion in an EPUB, or for formatting. HTML may also be used as the source document. HTML source documents may likewise be transformed into different HTML for delivery. The output HTML may include, for example: a new or updated Table of Contents; changed or additional metadata; or, for use with CSS Paged Media, additional copies of content to be removed from the flow and used in running headers and footers.

The first step in making accessible HTML is to use the most appropriate element whenever possible.

How HTML is presented in the browser, including dynamic content, user interface controls, and accessibility APIs, is out of scope for this paper. Important standards in this area include WAI-ARIA 1.1 [WAIARIA] and the rest of the WAI-ARIA 1.1 suite of specifications [WAIARIASUITE].

5. Tagged PDF

"Tagged PDF" is not a separate PDF specification. It refers to PDF that includes additional information about the logical structure of the document. Tagged PDF was first defined in PDF 1.4. Later versions of the PDF specification added more tag ('Structure Element') types and more properties of Structure Elements. PDF 2.0 [PDF2.0] added some new tags and deprecated some of the existing tags.

The text, graphics, and images in Tagged PDF can be extracted and reused for other purposes. For example, to make content accessible to users with visual impairments. PDF/UA files (see Section 6 [139]) are Tagged PDF files that also conform to additional requirements.

AH Formatter embeds PDF tags ('StructElem') for XSL Formatting Object elements as shown in Table 1 [137]. Other XSL-FO formatters have similar mappings.[1]

Table 1. XSL Formatting Objects and PDF tags

FO element	PDF 'Structure Element'	Comment
fo:root	Document	
fo:page-sequence	Part	
fo:flow	Sect	
fo:static-content	Sect	
fo:block	P or Div	P when it has the content of inline-level, otherwise Div
fo:block-container	Div or Sect	Sect when absolute-position="fixed" or "absolute", otherwise Div
fo:inline	Span or Reference	Reference when the child of fo:footnote, otherwise Span
fo:inline-container	Span	
fo:leader	Span	
fo:page-number	Span	
fo:page-number-citation	Span	
fo:page-number-citation-last	Span	
fo:scaling-value-citation	Span	
fo:index-page-citation-list	Span	
fo:bidi-override	Span	
fo:footnote		The footnote-reference-area embeds a Sect that contains all the footnotes on the page
fo:footnote-body	Note	
fo:float	Sect	
fo:external-graphic	Figure or Formula	Formula in case of MathML, otherwise Figure
fo:instream-foreign-object	Figure or Formula	Formula in case of MathML, otherwise Figure
fo:basic-link	Link	
itemizedlist	L	
listitem	LI	
listitem-label	Lbl	
listitem-body	LBody	
fo:table	Table	
fo:table-caption	Caption	
fo:table-header	THead	

[1]The information provided by other formatters, however, was either incomplete [FOP] or not in a format that could just be pasted into this paper [XEP].

FO element	PDF ‘Structure Element’	Comment
fo:table-footer	TFoot	
fo:table-body	TBody	
tr	TR	
td	TH or TD	TH within fo:table-header, otherwise TD
axf:form-field	Form	
axf:ruby	Ruby	
axf:ruby-base	RB	
axf:ruby-text	RT	

AH Formatter embeds PDF tags (‘StructElem’) for HTML/CSS elements and pseudo-elements as shown in the following table:

Table 2. HTML elements and PDF tags

HTML element	PDF ‘Structure Element’
html	Document
div	Div
h1	H1
h2	H2
h3	H3
h4	H4
h5	H5
h6	H6
p	P
ul	L
ol	L
li	LI
li::marker	Lbl
dl	L
dt	Lbl
dd	LBody
blockquote	BlockQuote
caption	Caption
table	Table
tr	TR
td	TD
th	TH
thead	THead
tfoot	TFoot
tbody	TBody
ruby	Ruby
rb	RB
rt	RT
span	Span
img	Figure
a[href]	Link
other block elements	Div
other inline elements	Span

5.1. Specialised PDF tags

When you generate PDF/UA or Tagged PDF output, the formatter maps each FO or each HTML element in the source to an equivalent ‘Structure Element’ in the PDF. However, there are several specialized Structure Elements that do not have correspondingly specialized FOs or HTML elements.

5.1.1. AH Formatter

In some cases, AH Formatter will map a general-purpose FO to the specialized Structure Element based on the context: for example, `fo:inline` ordinarily maps to 'Span', but the `fo:inline` child of `fo:footnote` maps to 'Reference' since the `fo:inline` is formatted to produce the footnote citation. Otherwise, to override the default mapping, specify the Structure Element name in the `axf:pdftag` property in XSL-FO or the `-ah-pdftag` property in CSS.

5.1.2. FOP

FOP implements a default mapping from FO type to Structure Element type, but the documentation [XEP] provides only three example mappings. Individual FOs may have the default mapping overridden using the `role` property.

5.1.3. XEP

XEP [XEP] provides two mechanisms for overriding its default mappings from FO to Structure Element name. The default mappings of each FO type to a Structure Element name may be overridden by specifying a custom `rolemap.xml` file. Individual FOs may have the default mapping overridden using the `rx:pdf-structure-tag` extension attribute. The allowed values are 'H1', 'H2', 'H3', 'H4', 'H5', 'H6', 'P' , 'TH', 'TR' and 'Artifact'. When the `rx:pdf-structure-tag` value is 'Artifact', an additional `rx:artifact-type` attribute may be specified. Its allowed values are 'Pagination', 'Page' and 'Layout'. When the `rx:artifact-type` value is 'Pagination', an additional `rx:artifact-type` attribute may be specified. Its allowed values are 'Header', 'Footer' and 'Watermark'.

6. PDF/UA

PDF/UA, defined in ISO 14289-1, is the specification intended for improving the accessibility of PDF based on the ISO 32000-1 (PDF 1.7) specification. ISO 14289-1 defines separate requirements for the PDF file format, behaviour of a conforming reader, and behaviour of Assistive Technology (AT) devices. Only the file format requirements are in scope for this paper.

The main features of PDF/UA file format requirements are:

- Contents must be tagged in logical reading order.
- Meaningful graphics, annotations and numerical formulas must include alternate text descriptions.

  Alternate text descriptions for graphics or numerical formulas can be specified by the `-ah-alttext` property, links can be specified by the '-ah-annotation-contents' property.
- Security settings must allow assistive technology to have access to the content.
- Including bookmarks in the PDF/UA is recommended.
- Annotations, links and multimedia may be included.
- The language of the document must be specified.
- All fonts must be embedded.

6.1. Matterhorn Protocol

The Matterhorn Protocol, published by the PDF Association, is a checklist of all the ways that it is possible for a PDF file to not conform to PDF/UA. The Matterhorn Protocol document consists of 31 Checkpoints comprised of 136 Failure Conditions. Some failure conditions can be checked programmatically, but others require human review.

Figure 1. Matterhorn Protocol failure conditions for tables

Checkpoint 15: Tables

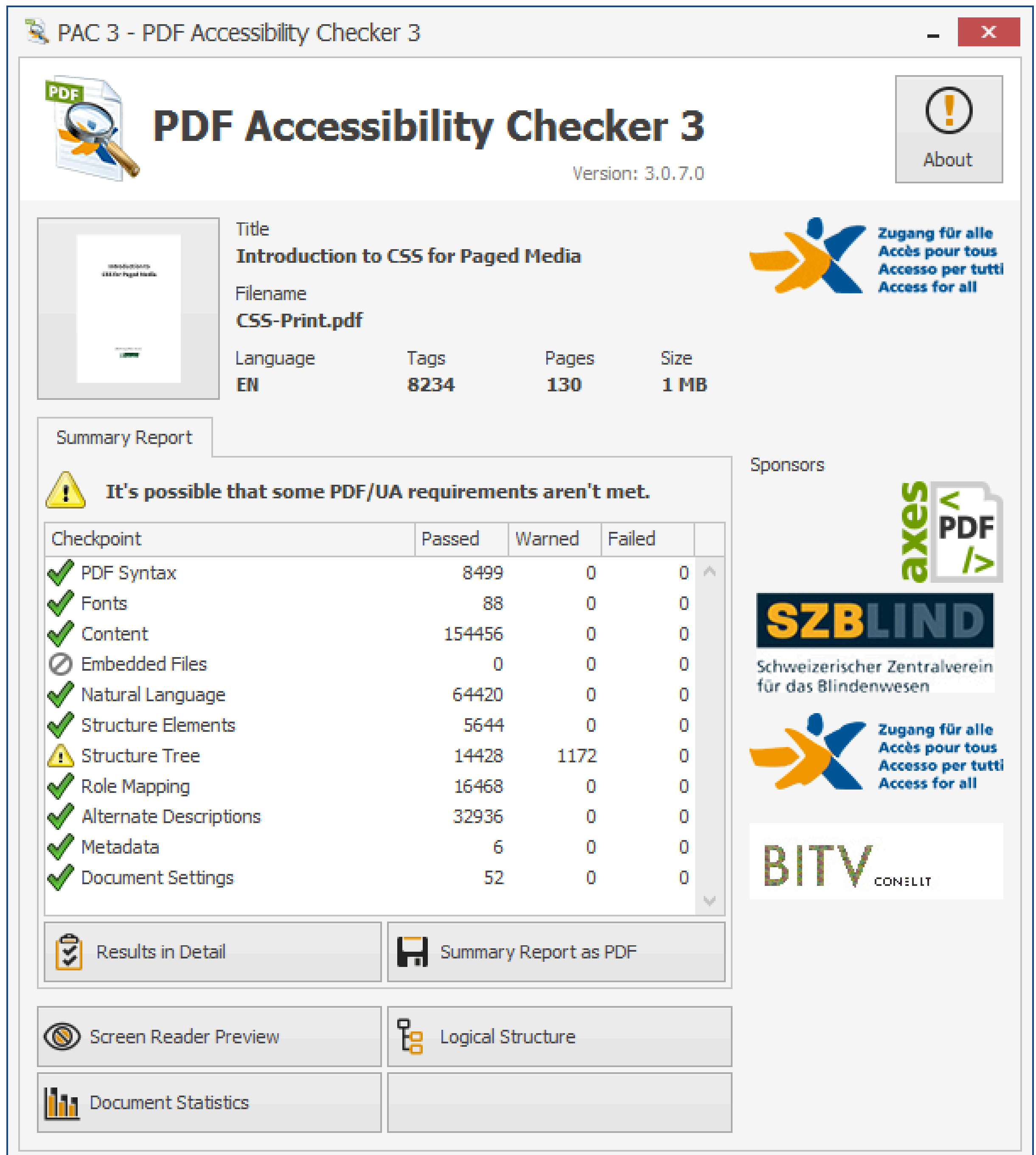
Index	Failure Condition	Section	Type	How	See
15 -001	A row has a header cell, but that header cell is not tagged as a header.	UA1:7.5-1	Object	Human	-
15 -002	A column has a header cell, but that header cell is not tagged as a header.	UA1:7.5-1	Object	Human	-
15 -003	In a table not organized with Headers attributes and IDs, a TH cell does not contain a Scope attribute.	UA1:7.5-2	Object	Machine	-
15 -004	Content is tagged as a table for information that is not organized in rows and columns.	UA1:7.5-3	Object	Human	-
15 -005	A given cell's header cannot be unambiguously determined.	UA1:7.5-2	Object	Human	01-006

6.2. PAC 3 PDF/UA checker

PDF Accessibility Checker 3 (PAC 3)[2] by Access For All is a freeware utility for Windows that checks PDF files for PDF/UA conformance. The program implements the Matterhorn Protocol checks. When you open a PDF file in PAC 3, the program runs its checks and shows a summary of the results. Since there is no interactive checking, the program can only warn about some of the failure conditions that require human checking.

[2] http://www.access-for-all.ch/en/pdf-lab/536-pdf-accessibility-checker-pac-3.html

Figure 2. PAC 3 PDF/UA checker



7. Common Structures

This section discusses how to represent structures that are commonly found in text documents as HTML, Tagged PDF, or PDF/UA (as applicable) for best accessibility.

7.1. Language indication

Correctly identifying the language assists assistive technologies.

7.1.1. HTTML

7.2. Part, Article, Section, or Division

There are multiple Structure Element types for representing the top-level hierarchy of the document:

Table 3. PDF tags for top-level structure of a document

Structure type	Description
Document	A complete document
Art	A relatively self-contained body of text with a single narrative. An article should not contain another article.
Sect	A container for grouping related elements
Div	A generic block-level element or group of elements

By default, AH Formatter maps `fo:root` to 'Document', `fo:page-sequence` to 'Part', `fo:flow` to 'Sect', and both `para-container` and any `para` that contains only block-level FOs to'Div'. It is possible, for example, to use `axf:pdftag` to map the higher-level `para-container` and `para` to 'Sect' if that better represents the structure of the document or to map `fo:root` to 'Art' if the document is a single article.

7.3. Headings

Headings are usually indicated by increasing the font size, font weight, or the space around the title, but these visual differences may not be apparent to a person using Assistive Technologies. When the document is 'strongly structured' (the document uses grouping elements such as 'Art', 'Sect', and 'Div' to represent the organization of the material) and the heading is always and only the first child of the grouping element, then use 'H' on the FO for the heading. When the document is not strongly structured, use the 'H1' to 'H6' Structure Elements to indicate heading levels.

When generating PDF/UA:

- Heading levels beyond 'H6' are allowed: for example, 'H7', and so on.
- A document may use either only 'H' Structure Elements or only 'H1' to 'H6' Structure Elements (and beyond).
- When using numbered heading levels:
  - The first heading must be 'H1'.
  - More than one instance of any heading level may be used.
  - There cannot be gaps in the descending heading level sequence: for example, 'H1' then 'H2' then 'H3' is a valid sequence, but 'H1' then 'H3' is not.
  - The heading level sequence can increment without restarting at 'H1' if that reflects the document structure. For example, 'H1', 'H2', 'H3', 'H2', 'H3', 'H4', 'H3' is a valid sequence.

7.4. Table of Contents

A Table of Contents is more likely to be generated on output than to be included in source XML. On the other hand, a HTML document that is not transformed before (or during) rendering may include the Table of Contents in the source document.

7.4.1. HTML

HTML does not include any semantic elements for a Table of Contents.

7.4.2. Tagged PDF

Use 'TOC' for a Table of Contents. The structure of a 'TOC' is similar to, but more flexible than, the structure of `fo:bookmark-tree`: a 'TOC' Structure Element may contain 'TOCI' (Table of Contents item) and 'TOC' entries. A 'TOCI' may contain any combination of 'Lbl', 'Reference', 'NonStruct', 'P', and 'TOC' Structure Elements.

Use 'TOC' for the block containing the Table of Contents. Use axf:pdftag="TOCI" for each entry in the Table of Contents. When the Table of Contents represents a hierarchy, use a block with 'TOC' for each level of the hierarchy.

Within each Table of Contents entry, use ‘Lbl’ for the title, ‘NonStruct’ for the leader or other content between the title and the page number, and use ‘Reference’ for the page number.

7.5. Index

Use ‘Index’ for the block containing the index.

7.6. Footnote

7.6.1. HTML

HTML does not have a semantic element for footnotes. [IDIOMS]

7.6.2. Tagged PDF

‘Note’is the inline-level Structure Element for explanatory text such as a footnote or an endnote. AH Formatter automatically tags `fo:footnote-body` as ‘Note’. However, ‘Note’ is defined in PDF 1.7 as an inline-level Structure Element, whereas an `fo:footnote-body` contains block-level FOs. As such, PDF/UA that is generated from XSL-FO that contains footnotes will be flagged by a PDF/UA checker.

Structure Elements for notes have been revised in PDF 2.0 []. PDF 2.0 has removed‘Note’ and added ‘FENote’, which is allowed at grouping, block, and inline levels.

7.7. Endnote

7.7.1. HTML

HTML does not have a semantic element for endnotes.

7.7.2. Tagged PDF

‘Note’is the inline-level Structure Element for explanatory text such as a footnote or an endnote. There is no FO specifically for endnotes, so use ‘Note’ for the endnote text.

7.8. Tables

Representing tabular data in XML is awkward enough without also adding accessibility features. Tables seem straightforward when they are just rows that each have the same number of columns, but real-world tables have cells that span rows and/or columns, table head rows, and table cells that act as headers for other cells in the same row or in following rows. Imagine trying to make sense of all of those when you can’t see the formatted table.

In this contrived example table (shamelessly borrowed from HTML 5.3 [TH]), the bold text (which you might not be able to see) acts as a header for cells below and/or to the right of the header cell:

Table 4. Measurement of legs and tails in Cats and English speakers

ID	Measurement	Average	Maximum
	Cats		
93	Legs	3.5	4
10	Tails English speakers	1	1
32	Legs	2.67	4
35	Tails	0.33	1

The following figure shows the relationships:

Figure 3. Scope in sample table

ID	Measurement	Average	Maximum
	Cats		
93	Legs	3.5	4
10	Tails	1	1
	English speakers		
32	Legs	2.67	4
35	Tails	0.33	1

7.8.1. HTML

HTML (since at least HTML 4.01) supports a 'scope' attribute for cells to which a header cell applies. The HTML 5 markup for the table is shown below:

```
<table>
  <caption>Measurement of legs and tails in Cats and English
speakers</caption>
  <thead>
    <tr> <th> ID <th> Measurement <th> Average <th> Maximum
  <tbody>
    <tr> <td> <th scope=rowgroup> Cats <td> <td>
    <tr> <td> 93 <th scope=row> Legs <td> 3.5 <td> 4
    <tr> <td> 10 <th scope=row> Tails <td> 1 <td> 1
  </tbody>
  <tbody>
    <tr> <td> <th scope=rowgroup> English speakers <td> <td>
    <tr> <td> 32 <th scope=row> Legs <td> 2.67 <td> 4
    <tr> <td> 35 <th scope=row> Tails <td> 0.33 <td> 1
  </tbody>
</table>
```

HTML also supports a 'headers' attribute for indicating the header cell or cells that apply to the current cell.

7.8.2. Tagged PDF and PDF/UA

In Tagged PDF, 'TH' structure elements may have a 'Scope' attribute, and both 'TH' and 'TD' structure elements may have a 'Headers' attribute. PDF/UA requires some features in a conforming PDF/UA file that are only optional in ordinary PDF. This includes 'Headers' attributes and, where 'Headers' attributes are not clear enough, also 'Scope' attributes.

The good news about accessible table markup is that any XML vocabulary that supports (X)HTML-compatible table markup supports 'scope' and 'headers' attributes. For example, 'scope' and 'headers' are in both JATS and DocBook as side-effects of their supporting HTML tables.

The not-so-good news about accessible table markup is that it requires human effort to properly markup the relationships between table cells, especially for convoluted tables. In practice, the attributes are poorly understood and are often

omitted. For example, the JATS 1.2 tag library currently omits 'scope' and 'headers' from its 'Accessibility' page, notes that 'scope' (though not 'headers') has not been widely supported, and inaccurately states that 'headers' points to rows and columns when it actually points to individual header cells.

7.9. Icons, etc.

7.9.1. Tagged PDF and PDF/UA

Use 'Artifact' for icons, etc., that do not represent meaningful content. Artifacts are "graphics objects that are not part of the author's original content but rather are generated by the conforming writer in the course of pagination, layout, or other strictly mechanical processes."[]

> **Note**
>
> NOTE: Artifacts may also be used to describe areas of the document where the author uses a graphical background, with the goal of enhancing the visual experience. In such a case, the background is not required for understanding the content. [PDF2.0]

Use 'NonStruct' for any additional FOs that are necessary only to generate the correct visual appearance of a note, etc. A 'NonStruct' Structural Element is not interpreted or exported to other document formats but its descendants are processed normally.

7.10. Mathematics

7.10.1. HTML

HTML 5 supports MathML, but MathJax [MATHJAX] is an increasingly popular JavaScript library for displaying mathematics as a graphic. HTML 5 does not have a standard way to indicate that a graphic represents mathematics. MathJax will, for example, include role="math" in generated SVG but in mathematics that is output as HTML markup.

7.10.2. Tagged PDF

PDF 1.7 defines both 'Figure' and 'Formula' Structure Element types for illustrations (as well as 'Formula' for interactive forms).

AH Formula supports MathML 3, and it automatically tags an `fo:external-graphic` or `fo:instream-foreign-object` that contais MathML as 'Formula' and tags every other `fo:external-graphic` or `fo:instream-foreign-object` as 'Figure'. It is not known what other formatters generate for MathML.

If you have to use a graphic, and not use MathML, to represent mathematics, then use 'Formula' to identify the graphic as a formula. You should also use `axf:alttext` (or equivalents) to provide alternate text for the image.

7.11. Citation

7.11.1. HTML

HTML 5 provides the `cite` element for denoting the title of a work. The `blockquote` and `q` elements both have a `cite` that is a link to the source of the quotation.

7.11.2. Tagged PDF

Use 'Reference' for a citation to content that is elsewhere in the document.

7.12. Block quotation

7.12.1. HTML

HTML 5 provides the `blockquote` and `q` element.

7.12.2. Tagged PDF

Use 'BlockQuote' to tag a block quotation. While a block quotation typically has a different appearance to the surrounding text, the visual differences may not be apparent to a person using Assistive Technologies.

> Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam bibendum tincidunt pharetra. Aenean ultricies molestie ante, sit amet ultricies nunc mollis id. Integer ut porttitor felis, vel tincidunt velit. Duis volutpat, quam quis aliquet tristique, nulla dui malesuada velit, et consectetur tellus ipsum et arcu.

7.13. Inline quotation

Use 'Quote' to tag a quotation, such as "“Lorem ipsum dolor sit amet”".

7.14. Inline code

Use 'Code' to tag a fragment of computer program text, such as "`axf:pdftag="Code"`".

7.15. Bibliography

There is no Structure Element for a bibliography. However, use 'BibEntry' for an entry in a bibliography. The 'BibEntry' may contain a 'Lbl' Structure Element, but there are no Structure Element types for the parts of a bibliography entry, such as author, publisher, publication date, and so on.

8. Conclusion

Accessibility does matter. There are myriad details that can mastered. Software can try to "do the right thing" when processing or formatting your data, but it is better, and it will yield a better result, if the effort is made to include accessibility information in your source XML.

Bibliography

[Ang] Trish Ang: How to Fail at Accessibility, 22 February 2019, Slack. https://web.archive.org/web/20190426183254/https://slack.engineering/how-to-fail-at-accessibility-99bdf3504f19

[FOP] Apache™ FOP: Accessibility The Apache Software Foundation, https://web.archive.org/web/20190522213553/https://xmlgraphics.apache.org/fop/2.3/accessibility.html

[GIGO] *GIGO* in The Jargon File, Yash Tulsyan, https://web.archive.org/web/20130827121341/http://cosman246.com/jargon.html#GIGO

[IDIOMS] *Common idioms without dedicated elements* in HTML 5.3, Editor's Draft, 18 October 2018, W3C, https://web.archive.org/web/20181124235038/http://w3c.github.io/html/common-idioms-without-dedicated-elements.html

[JATS] Journal Publishing Tag Library NISO JATS Version 1.2 (ANSI/NISO Z39.96-2019), May 2019, National Center for Biotechnology Information (NCBI), National Library of Medicine (NLM), https://web.archive.org/web/20190528113513/https://jats.nlm.nih.gov/publishing/tag-library/1.2/index.html

[MATHJAX] MathJax, May 2019, MathJax Consortium, https://web.archive.org/web/20190527040208/https://www.mathjax.org/

[Matterhorn] Matterhorn Protocol, PDF Association, https://www.pdfa.org/publication/the-matterhorn-protocol-1-02

[PDF1.7] ISO 32000-1:2008, Document management — Portable document format — Part 1: PDF 1.7

[PDF2.0] ISO 32000-2:2017, Document management — Portable document format — Part 2: PDF 2.0

[Perera] Chandi Perera: Accessible Publishing: What is it and how do we get there?, 22 March 2018, Typefi. https://web.archive.org/web/20190527111115/https://www.typefi.com/typefi-user-conference-2018/presentations-2018/what-is-accessible-publishing/

[Rosenblum] Bruce Resonblum: @scope and @headers, 9 April 2019, Inera. https://web.archive.org/web/20190527120216/https://www.biglist.com/lists/lists.mulberrytech.com/jats-list/archives/201904/msg00004.html

[TaggedPDF] *Tagged PDF* in PDF Output, Antenna House, Inc., https://web.archive.org/web/20190527124525/https://www.antennahouse.com/product/ahf66/ahf-pdf.html

[TH] *The th element* in HTML 5.3, Editors Draft, https://web.archive.org/web/20181020164617/https://w3c.github.io/html/tabular-data.html#the-th-element

[WAIARIA] Accessible Rich Internet Applications (WAI-ARIA) 1.1, 14 December 2017, W3C https://www.w3.org/TR/wai-aria-1.1/

[WAIARIASUITE] WAI-ARIA Overview, 15 January 2016, W3C https://web.archive.org/web/20190527024541/http://www.w3.org/WAI/standards-guidelines/aria/#wai-aria-1_1

[WCAG2.1] Web Content Accessibility Guidelines (WCAG) 2.1, 5 June 2018, W3C https://www.w3.org/TR/WCAG21/

[XEP] XEP User Guide, RenderX https://web.archive.org/web/20181225121734/http://www.renderx.com/reference.html

Scrap the App, Keep the Data

Barnabas Davoti

Abstract

Whether by reacting swiftly to shifting market conditions and disruptive technologies or by growing through mergers and acquisitions, the ability to adapt is critical to success in the digital age.

As business processes and workflows evolve, new technologies and systems are constantly introduced to support them. Over time, IT estates become fragmented, and the number of legacy systems increases year by year. Valuable data ends up in isolated silos, only accessible via otherwise unnecessary applications that are expensive to maintain and license or are approaching the end of life.

This paper suggests a generic approach to transform arbitrary relational data into aggregated, hierarchical form and build a thin application to provide read access to end users.

In my presentation, I would like to point out why the relational model falls short when it comes to data aggregation.

XML can easily model both relational and hierarchical data. It's an excellent choice for data re-modeling and also for building a thin, data access application.

The approach is based on experience from multiple projects.

XML "*bricks*" used in the solution:

- processing pipeline configuration as XML (Apache Ant)
- XSLT
- XQuery
- XML database

1. Business issue

Quite often applications are built to be the only gate to its data. Business requirements change all the time, and when the app cannot adapt to a new workflow, or for other reasons becomes obsolete, it's a bottleneck.

Read-only access to the data is still essential for the business, but they cannot justify keeping, maintaining the app.

A possible solution is to replace the complex data-creator app with a simple, thin, data-access app.

The approach I'm sketching here is based on experience from multiple projects. Also, some projects were so massive that we had to work with vastly diverse data. These circumstances pushed us to think generic.

Let me explain this through example use cases.

1.1. Data archiving and application decommissioning

It is the most common use case. For example, a company changes the financial software because of an acquisition. According to the regulations, they still have to keep its data for a few years.

It's historical data that won't change anymore, however, occasionally it has to be accessible to end users as a reference, also for auditors.

For this use case, the natural choice for backend and user interface is an archiving solution. It usually provides a wizard to configure a search application GUI and implements the archive administration workflow (user roles, retention policy, etc).

However, the archiving system requires data packages with aggregated records, which is a challenge.

1.2. Freeing up licenses

A corporation pays for expensive PLM (product lifecycle management) software licenses for hundreds of users. However, only a handful of users change any product data in the system; the rest use it as a read-only reference.

The company decides to keep only a few licenses and migrate the data nightly into a database which gives read-only access. The new, thin app could also provide a more easy to use, intuitive GUI for the less demanding user.

1.3. Merging data silos

In this scenario, multiple applications from different vendors are used in one company workflow. The applications are not integrated, and their data is isolated, only accessible by a single application.

From the user's perspective, the data across these databases should be connected. Switching between these apps all the time makes the work inefficient, and is a source of frustration for the users.

If there are distinct user roles for data creation and data access, it could be highly beneficial for the data access user to see all the data related, harmonized, and accessible via a single GUI.

Finding the right connection points between the silos can be challenging, but if there are some permanent global identifiers, which are not vendor/system specific, then those can be used to make the connections, aggregate the data across the isolated silos.

2. Technical angle

Our ultimate goal is to provide access to the data via a thin app, *as simple as possible*. With other words: *as cheap as possible* to implement, or perhaps even generate it via a wizard, no coding needed.

When a user wants to access information via a traditional application with a relational database backend, it does two things in real time:

1. Filters records

2. Aggregates data

The data always have to be aggregated for human consumption. The only question is when we do this:

1. In real time - when the user queries the data, OR
2. In advance - so we store the data aggregated

Aggregated data is usually redundant, and a data creator application can't afford this. The relational data model is efficient because it does not store data redundantly. However, on-demand data aggregation can be very complex and resource-intensive.

Can we persist the data in an aggregated form, so the application can be simplified and only needs to deal with record filtering? The relational model falls short on this. It can do aggregation partly, but not entirely. The hierarchical model is a better choice.

2.1. Relational data

Let's look into this via an example. We want to build a thin app which gives access to travel reports. Our aggregated data record - what the thin app's search interface will filter on - is a *report*, which has a header (travel date, reason, employee info, etc.) and a list of transactions (cost items).

The following simple tables hold data about a travel report.

Table 1. Travel table - holds one record per trip

travel id	employee id	date	reason
t1	e1	2019-05-15	Customer meeting in Oslo

Table 2. Employee table - one row per employee

employee id	first name	second name
e1	Ola	Nordmann

Table 3. Transaction table - one row describes one transaction

transaction id	travel id	item name	cost	currency
c1	t1	bus ticket	100	NOK
c2	t1	accommodation	1000	NOK

Can we make an aggregate table where a single record holds the whole report? Only partly.

Table 4. Aggregate table

travel id	empl.id	first name	second name	date	reason	trans id	item name	cost	currency
t1	e1	Ola	Nordmann	2019-05-15	Customer meeting in Oslo	c1	bus ticket	100	NOK
t1	e1	Ola	Nordmann	2019-05-15	Customer meeting in Oslo	c2	accomodation	1000	NOK

The aggregate table is highly redundant; still, it does not let us add all report data into a single record. Any application that uses this as a source will have to do further aggregation beyond record filtering.

2.2. Hierarchical data

Example 1. Aggregated record as a hierarchy:

- travel id=t1
  - date: 2019-05-15
  - reason: Customer meeting in Oslo
  - employee name: Ola Nordmann

- transactions
  - bus ticket - 100 NOK
  - accommodation - 1000 NOK

Example 2. Serialized as XML:

```
<travel id="t1">
   <date>2019-05-15</date>
   <reason>Customer meeting in Oslo</reason>
   <employee id="e1">
       <first-name>Ola</first-name>
       <second-name>Nordmann</second-name>
   </employee>
   <transactions>
       <transaction id="c1">
           <item-name>bus ticket</item-name>
           <cost currency="NOK">100</cost>
       </transaction>
       <transaction id="c2">
           <item-name>accommodation</item-name>
           <cost currency="NOK">1000</cost>
       </transaction>
   </transactions>
</travel>
```

A hierarchical (XML) database can store records like this, and a thin app then should only filter the records relevant to the user and render it.

3. Different paths to consider

Depending on data complexity and the availability of the different options, we need to choose the best path from the list below.

3.1. Use a proprietary connector

This is the fastest and safest way. For example, SAP has a connector for the archiving software what we used, but it only covered 10% of all the scenarios in our projects.

3.2. Use the app's export/import functionality

Use the data creator app's high-level export/report functionality - one last time if the application will be decommissioned - to get all the data out aggregated. It's usually a reliable way, assuming it's a mature product.

Then we get

1. Aggregated and structured data

   Likely we need to transform this further to be able present this to the user, but it's already aggregated, so it's easy.

2. Aggregated and unstructured data

   The report tool could produce PDFs, which is human-readable, but searchability is terrible. We need at least some metadata on top of the documents.

3.3. Export the data from the app's relational database and aggregate it into hierarchical XML records

Timing - The archiving scenario: at the point of time of an app decommissioning, we could likely still find people in the company who know the data. Maybe that knowledge will disappear later on, so doing the aggregation as early as possible is an advantage.

Complexity - It's a low-level approach, so a reasonable level of understanding the database schema is necessary. It depends on the application, of course. Sometimes it's also possible to work with *"aggregation tables"*, which at least partly aggregates the data in the relational database, which makes our job easier.

Example 3. Travel reports:

In one project, we had to aggregate and archive travel records - similar to the example I used earlier in this paper. It has a header (date, employee...) and transaction items in the body.

It's quite simple on the database schema level:

1. Main travel table - one record per trip.
2. Transaction table - one record per transaction.
3. Attachment table - one record with the attachment metadata and a column with content, for ex. PDF, stored as BLOB.
4. Approx. a dozen other tables were *"register tables"* . For example, based on the employee ID we could look up the employee's name from another table.

All in all, we used about 15 tables, and aggregated the records with XQuery that was not so complicated, less than 100 lines and the same patterns were repeating. More about this later.

3.4. Export the data from the app's relational database, serialize and store it as relational XML records

In this scenario, we skip the aggregation step. We serialize the relational tables "*as-is*" as XML during the export process without remodeling the data.

Maybe, in theory, this scenario does not make too much sense, but is quite often applied in case of a data archive. Simply because the archive application provides the thin app out of the box, and also the data management workflow.

Then for fetching the relevant info, we have to do two things on demand:

1. filter the records
2. aggregate the data

So we need to write almost the same aggregator XQuery what we use in the previous scenario, but this is run on-demand by the end-user in real-time and not in advance as part of the data migration.

Pros:

- more flexible than the pre-aggregated

Cons:

- could lead to bad performance
- more complex application
- the relational model know-how might disappear from the corporation in the meantime

3.5. Data virtualization

With a generic data virtualization tool, it's also possible to query a relational database via XQuery and build aggregated, hierarchical XML directly. It can provide a reliable platform and decrease the complexity of the data aggregation. At the time we worked on these projects, I was not aware of this option.

4. Implementation

In this paper, we'll focus on exporting the data from the app's relational database and aggregating it into hierarchical XML records.

Our data pipeline:

- source: arbitrary relational database
- target: XML database with aggregated, hierarchical records

We could use another hierarchical format, like JSON, but the XML platform is mature and gives us all that we need: serialization format, schema-, transformation- and query language, and also a native database for persistence. (I might be partial.)

The suggested solution does use some lines of Java code, but it's kept to the minimum and prefers the XML technology stack.

4.1. Execution framework

The data pipeline is based on Apache Ant (open source). Its main configuration is XML-based.

It might count deprecated as a build tool, but it still shines as a process execution framework with rich standard features, like:

1. Process execution (command line, Java, etc.)
2. File operations
3. XML operations (XSLT execution, schema validation, etc)

Beyond the standard features, also 3rd party tools integrate with Apache Ant, like:

- DBUnit, the tool we used to export data from relational databases
- An archiving system we used during our projects

4.2. Execution steps

4.2.1. Export

The export process fetches metadata and data from a relational database and persists it as XML files.

It does the following:

1. Extracts database metadata information (table- and columns names, types, etc.) using Java code (JDBC) and serializes it as XML.
2. Uses this metadata XML as an input and creates an Ant export configuration for DBUnit. Code: XSLT.
3. Runs DBUnit to export the data and serializes it as XML (relational model), since - as mentioned earlier - DBUnit has Apache Ant integration. Code: XML.

DBUnit is an open source tool to export and import database data to and from XML datasets. It's database vendor agnostic, has a connector for most widely used databases. It has a streaming mode to deal with big data. In our configuration, we exported each table into a single XML file. The largest XML output file so far was 43 GB.

Example 4. XML data using the relational model - from DBUnit:

```
<table name="employee">
    <column>employee-id</column>
    <column>first-name</column>
    <column>second-name</column>
    <row>
        <value>e1</value>
        <value>Ola</value>
        <value>Nordmann</value>
    </row>
    <row>...</row>
```

```
</table>
```

4.2.2. Transformation

4.2.2.1. XML serialization

Serialization of any data into XML has its challenges. First of all, DBUnit has to figure out the type of the data field. Is it:

- text
- numeric
- binary

If it's binary, then it gets embedded into XML as a Base64 encoded string. All the 64 characters this encoding uses are valid XML characters.

Serializing numeric values is easy. However, text data can contain characters which are invalid in XML. For example, some legacy applications used "*control characters*" - below decimal 32 (space char). Most of those are invalid in XML 1.0. XML 1.1 expands the set of allowed characters.

Of course, we could work around this by Base64 encoding all textual data, but this would result in cumbersome processes when we query the data.

In projects, we did stick with XML 1.0, and when we faced illegal XML characters in the DBUnit output, mostly we could simply remove them safely with a low level (text) process.

4.2.2.2. Data types

Getting the data types converted correctly is a challenging part of the process. SQL implementations bring vendor-specific data types. All of them have to be mapped accurately to XML schema data types. This is done via configuration.

4.2.2.3. Row customizer

In our framework, we tried to implement the process to be as generic as possible, but we also added customization options.

Our row customizer is a custom XQuery that gets a single row as input and can filter or modify it.

4.2.2.4. Format conversion

The serialization format DBUnit uses is not the best when we want to query the data set. It's worth to write generic code to turn XML with column names as text, into XML with column names used as element names.

Example 5. XML data using the relational model - better semantics:

```
<employee>
    <row>
        <employee-id>e1</employee-id>
        <first-name>Ola</first-name>
        <second-name>Nordmann</second-name>
    </row>
</employee>
```

4.2.3. Aggregation

The most challenging step in this data flow is to re-model the relational data into aggregated, hierarchical data. A relational database table becomes one (or more) XML document during the data export, which can be quite large, often several gigabytes. During the aggregation, we need to "*join*" these large "*table documents*".

We can use SAX, StAX or XSLT streaming mode to process large files, but the parallel processing of several large documents is problematic. Here comes the power of an XML database and XQuery. Configuring proper indexes is a must.

As I have described before, the complexity of the XQuery is not too bad, since we tend to reuse the same patterns multiple times.

This XQuery is specific to the database schema we work with, and cannot be written generically. This query script is the only part of the data pipeline which is custom made for each relational database schema; the other operations are generic and work with any database.

Most importantly, we need to have a decent knowledge of the relational database schema, and also the end user requirements about how this data will be searched and used.

Example 6. Aggregated records in hierarchical XML:

```
<travels>
    <travel id="t1">
        <date>2019-05-15</date>
        <reason>Customer meeting in Oslo</reason>
        <employee id="e1">
            <first-name>Ola</first-name>
            <second-name>Nordmann</second-name>
        </employee>
        <transactions>
            <transaction id="c1">
               <item-name>bus ticket</item-name>
               <cost currency="NOK">100</cost>
            </transaction>
            <transaction id="c2">
                <item-name>accommodation</item-name>
                <cost currency="NOK">1000</cost>
            </transaction>
        </transactions>
    </travel>
    <travel>...</travel>
    ...
</travels>
```

It's also possible that the data from one database is not aggregated into a single list of records, but into multiple lists of records using different XML schemas.

5. Conclusion

We had great focus to make this solution as generic as possible to

1. optimize productivity
2. provide good control on the process

The first statement is likely trivial, while the second requires a bit more consideration. It's quite challenging to test complex data migration processes. We can count records, but if we want to compare data, that's a demanding task. The easiest way to ensure quality is to build a robust, generic framework, which has only a few "*moving parts*", code which is database schema specific.

When the data aggregation is done, we either use an application platform or archiving system to create a thin app via an app wizard, or we build it on top of the (XML) database using XQuery.

If we feed an archiving system with aggregated, hierarchical records, it can provide the complete end-user application via simple configuration: the end user GUI (search forms), the result rendering, and even the record filtering query (for example XQuery) can be auto-generated.

Documenting XML Structures

Erik Siegel

May 2019

1. Introduction

We, as XML geeks, all have to do with understanding XML structures we didn't design ourselves. It might be a programming language written as XML, a library configuration file we need to change or data that needs filling. Whatever it is, we need to comprehend the format: the elements, the parent-child relations and the attributes. Most important of course is the *meaning* of it: how can we use this XML structure to bridge the gap between our intend and the workings of the software it's going to be processed by.

Most of us will probably have been on the other side of this problem: We designed a nifty XML structure and needed to explain this in such a way that others can and will use it (and, as an important side-effect, are mighty impressed by our elegant, intelligent and beautiful design). How to explain an XML structure so users can make the most of it? More often than not this also has marketing value: A well constructed and intelligible explanation might convince people to use your product/library/programming language.

Neither side is easy. We probably all have experienced the frustration of wrestling with a half understood and sloppily documented XML format as input for some piece of software that stubbornly refused to do what we wanted it to do. Or, on the other side, the sinking feeling of having to document this nice but complex XML structure, not really knowing how to do this, but realizing it will take a lot of time.

This paper will look at the problem of documenting XML structures from various angles: consumer, producer, the structures themselves, supporting software, etc. It will then focus on the production side. Spoiler alert: There are no quick and easy solutions and at the end there is, unfortunately, no GitHub repository containing an auto-XML-structure-document-writer-application... sorry. However, with a little thinking upfront and a bit of automation, the task of documenting XML structures can become more manageable and maybe, for some, even enjoyable.

1.1. About the author and his documenting experience

I'm an XML specialist, doing things like consulting, designing and programming, strictly XML technology only. However, strangely enough, I also like documenting, writing and explaining things to people. Some examples:

- I have given various XML, XSLT and other related courses to very different audiences.
- Together with Adam Retter I wrote a book about eXist-db, which was published by O'Reilly in 2014.
- Some time ago I re-factored the eXist documentation pages (and partly the accompanying application).
- I'm currently working on a book about XProc 3.0, to be published when we've finished the standard.
- Of course, like probably for everyone, documenting stuff is something I sometimes do as part of the project I'm working on.

In all this, documenting XML structures is unavoidable. So I've tried out different ways and formats and tried to understand why some things work and others don't. This culminated into this (admittedly not very academic but hopefully enlightening) paper.

2. Consumption: Understanding XML structures

If we in our line of work encounter an XML structure we don't know yet, finding some description is usually just a web-search away. Google, Stackoverflow and the likes will often come up with some example, solution or description to help us. This is what I call "cookbook" level understanding: You don't really know what you're doing but you found a recipe and, hey, it works (or not).

But what if you really need to understand the format and its implications? For instance because it's a programming language you need to master. Or the application it serves is important to you. You start digging and hopefully find some kind of in-depth explanation of the structure. Do this more than once and you'll find there is no uniform way of documenting such a thing. Let's have a look at a few:

XSLT
: A lot of people in the XML community know Michael Kay's XSLT book (Michael Kay; XSLT 2.0 and XPath 2.0 Programmer's Reference, 4th edition; Wiley Publishing). Since an XSLT program is an XML structure, the book needs to explain this. Here's an excerpt from the explanation of the `<xsl:copy-of>`:

Figure 1. An excerpt from the documentation of XSLT

xsl:copy-of

The main purpose of the `<xsl:copy-of>` instruction is to copy a sequence of nodes to the result sequence. This is a deep copy — when a node is copied, its descendants are also copied.

Changes in 2.0

A new `copy-namespaces` attribute is introduced: This gives you control over whether or not the unused namespaces of an element should be copied.

Two new attributes `validation` and `type` are available to control whether and how the copied nodes are validated against a schema.

Format

```
<xsl:copy-of
  select = expression
  copy-namespaces? = "yes" | "no"
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname />
```

Position

`<xsl:copy-of>` is an instruction. It is always used within a sequence constructor.

Attributes

Name	Value	Meaning
select mandatory	XPath Expression	The sequence of nodes or atomic values to be copied to the output destination
copy-namespaces optional	«yes» or «no». Default is «yes»	Indicates whether the namespace nodes of an element should be copied
validation optional	«strict», «lax», «preserve», or «skip»	Indicates whether and how the copied nodes should be subjected to schema validation, or whether existing type annotations should be retained or removed
type optional	Lexical QName	Identifies a type declaration (either a built-in type, or a user-defined type imported from a schema) against which copied nodes are to be validated

The `type` and `validation` attributes are mutually exclusive: if one is present, the other must be absent. These attributes are available only with a schema-aware processor.

Content

None; the element is always empty.

All XML structure descriptions look this way, with as most important parts: short introduction, a standardized formatted impression of what it looks like, a table explaining the attributes and a description of its contents.

Maven POM

The documentation of Maven POM (Project Object Model) files can be found on the Apache Maven POM site (https://maven.apache.org/pom.html). Here's an example:

Figure 2. An excerpt from the documentation of Maven POM files

The BaseBuild Element Set

`BaseBuild` is exactly as it sounds: the base set of elements between the two `build` elements in the POM.

```
<build>
  <defaultGoal>install</defaultGoal>
  <directory>${basedir}/target</directory>
  <finalName>${artifactId}-${version}</finalName>
  <filters>
    <filter>filters/filter1.properties</filter>
  </filters>
  ...
</build>
```

- **defaultGoal**: the default goal or phase to execute if none is given. If a goal is given, it should be defined as it is in the command line (such as `jar:jar`). The same goes for if a phase is defined (such as install).
- **directory**: This is the directory where the build will dump its files or, in Maven parlance, the build's target. It aptly defaults to `${basedir}/target`.
- **finalName**: This is the name of the bundled project when it is finally built (sans the file extension, for example: `my-project-1.0.jar`). It defaults to `${artifactId}-${version}`. The term "finalName" is kind of a misnomer, however, as plugins that build the bundled project have every right to ignore/modify this name (but they usually do not). For example, if the `maven-jar-plugin` is configured to give a jar a `classifier` of `test`, then the actual jar defined above will be built as `my-project-1.0-test.jar`.
- **filter**: Defines `*.properties` files that contain a list of properties that apply to resources which accept their settings (covered below). In other words, the "`name=value`" pairs defined within the filter files replace `${name}` strings within resources on build. The example above defines the `filter1.properties` file under the `filters/` directory. Maven's default filter directory is `${basedir}/src/main/filters/`.

  For a more comprehensive look at what filters are and what they can do, take a look at the quick start guide.

This follows a different approach: Short introduction, example XML fragment, explanatory text with a lot of bullets.

Ant

Another example we're probably all familiar with is Ant. Ant's documentation can be found on the Apache Ant site (https://ant.apache.org/manual/). Building blocks of Ant are tasks. Here's an example of a task description:

Figure 3. An excerpt from the documentation of Ant

Echo

Description

Echoes a message to the current loggers and listeners which means `System.out` unless overridden. A *level* can be specified, which controls at what logging level the message is filtered at.

The task can also echo to a file, in which case the option to append rather than overwrite the file is available, and the *level* option is ignored

Parameters

Attribute	Description	Required
message	the message to echo.	No; defaults to a blank line unless text is included in a character section within this element
file	the file to write the message to.	No; only one of these may be used
output	the Resource to write the message to (see note). *Since Apache Ant 1.8*	
append	Append to an existing file (or open a new file / overwrite an existing file)?	No; ignored unless *output* indicates a filesystem destination, default is `"false"`
level	Control the level at which this message is reported. One of `"error"`, `"warning"`, `"info"`, `"verbose"`, `"debug"` (decreasing order)	No; default is `"warning"`
encoding	encoding to use. *since Ant 1.7*	No; defaults to default JVM character encoding
force	Overwrite read-only destination files. *since Ant 1.8.2*	No; defaults to `"false"`

Examples

Basic use:

```
<echo message="Hello, world"/>
```

```
<echo message="Embed a line break:${line.separator}"/>
```

The documentation follows a consistent pattern: Description, parameters (which are attributes), description of nested elements (for brevity reasons not shown) and examples.

HL7 CDA

Here's an example of something only a few will be familiar with: HL7 CDA (health Level 7, Clinical Document Architecture), a standard for the exchange of healthcare information. In a book that tries to explain this and more (Principles of Health Interoperability, SNOMED CT, HL7 and FHIR; Tim Benson, Grahame Grieve; Springer-Verlag), I found the following way of explaining the CDA XML structure:

Figure 4. An excerpt from the documentation of HL7 CDA

Body

Every CDA document has one header and one body part (see Fig. 15.3).

Fig. 15.3 CDA body

The body is either a <nonXMLBody> or a <structuredBody>. <nonXMLBody> is present to provide upward compatibility with CDA Level 1, and may contain any type of human-readable data including text (txt, rtf, html or pdf) or image (gif, jpeg, png, tiff or g3fax). Data encoded using XML may not be put in the <nonXMLBody>.

```
<component>
<nonXMLBody mimeType="text/plain">
<text> Text goes here </text>
</nonXMLBody>
</component>
```

<structuredBody> is used for XML-encoded data. It is the root node for one or more < section >.

```
<component>
<structuredBody>
<section>
<code code="xxxx" codeSystem="xxxx"
codeSystemName="xxxx" displayName="xxxx"/>/>
<title>xxxx</title>
<text>
xxxx<content styleCode="xxxx">xxxx</content>xxxx
</text>
</section>
```

They use a tree diagram to outline the structure followed by a short explanation followed by (badly formatted…) examples of XML structures and more text to explain these.

eXist-db `conf.xml`

A last example comes from the documentation of eXist-db's `conf.xml` file. This is the main configuration file for the database.

Figure 5. An excerpt from the documentation of eXist-db's **conf.xml**

```
<!--
    Settings for the database connection pool:

    - min:
        minimum number of connections to keep alive.

    - max:
        maximum number of connections allowed.

    - sync-period:
        defines how often the database will flush its
        internal buffers to disk. The sync thread will interrupt
        normal database operation after the specified number of
        milliseconds and write all dirty pages to disk.

    - wait-before-shutdown:
        defines how long the database instance will wait for running
        operations to complete before it forces a shutdown. Forcing
        a shutdown may leave the db in an unclean state and may
        trigger a recovery run on restart.

        Setting wait-before-shutdown="-1" means that the server will
        wait for all threads to return, no matter how long it takes.
        No thread will be killed.
-->
<pool max="20" min="1" sync-period="120000" wait-before-shutdown="120000"/>
```

This file has no separate documentation page or document. Instead everything is documented using XML comments inside the file itself.

These are just a few examples and they illustrate there are very different of documentation formats out there. Now everybody probably has his/her preferences and not every format needs the same kind or level of documentation, so can we say something in general about it? Let's try:

- A major factor in the comprehensibility of the descriptions is *consistency*. All parts of the structures must be explained in the same way, using the same (sub)sections and lay-out. This becomes more and more important when you use the XML structure on a regular basis. Your eyes and brain get used to the format and can quickly and easily find the things you want to know.
- The structure description must be correct and complete. No unmentioned surprise attributes for special occasions, no missing discrete value lists, etc.
- On the other hand: Constructions or values that will rarely be used should be recognizable as such. You don't want to spend time understanding this weird attribute that seems important, finding out it will only rarely be used, much later...
- The format should be easy to interpret. Meaning should almost pop-up from the page by just looking at it. This means judicious use of all the lay-out tools we have: sections, tables, colors, fonts, whatever. An *attractive* look is almost a necessity (you don't get a second chance for a fist impression).

3. Production: Creating XML Structure documentation

Let's step to the other side and imagine we have to document some non-trivial XML structure. If there are no previously set rules and guidelines, we're faced with some tough choices: how deep or shallow should the documentation be, what format are we're going to use, how are we going to produce and maintain it?

These questions need in answer, but maybe we should start with acknowledging something important: for most people *it's not a pleasant prospect at all!*. You're deep-down in some code that will be used by others, you created some clever XML format, and suddenly you realize that, for people to be able to use your skilfully designed contraption, this format, that grew a little out of hand, needs documentation. Argh. A depressive sinking feeling overwhelms you... But why is that?

- Creating documentation means stepping out of your "knowledge bubble", which is a very, very hard thing to do. You have to step back and try to imagine you're new to all this. What does a fresh user of your software/format needs to know? What is important background, overview and detail information?

- Creating documentation is a lot of work.
- You're (probably) a developer, not a technical writer, and documenting is not your favourite cup of tea.
- Even when you just start writing, you'll soon realize you need a *format*, a consistent way of describing things. Not only for the reader, but also for yourself. Documenting structures by following some format is really much easier than just typing along and making things up on the fly. But which one? Sigh...
-
- Once you have it, you're obliged to *maintain* it. This means that changes in the software now also require you to revisit the documentation. This can lead to all the phenomena we know from maintaining software: from just a few keystrokes to extensive refactor and overhaul operations.
- Sometimes, while programming, you get carried away and in the end your XML structure turns out to be so complicated, documenting it is hardly doable.

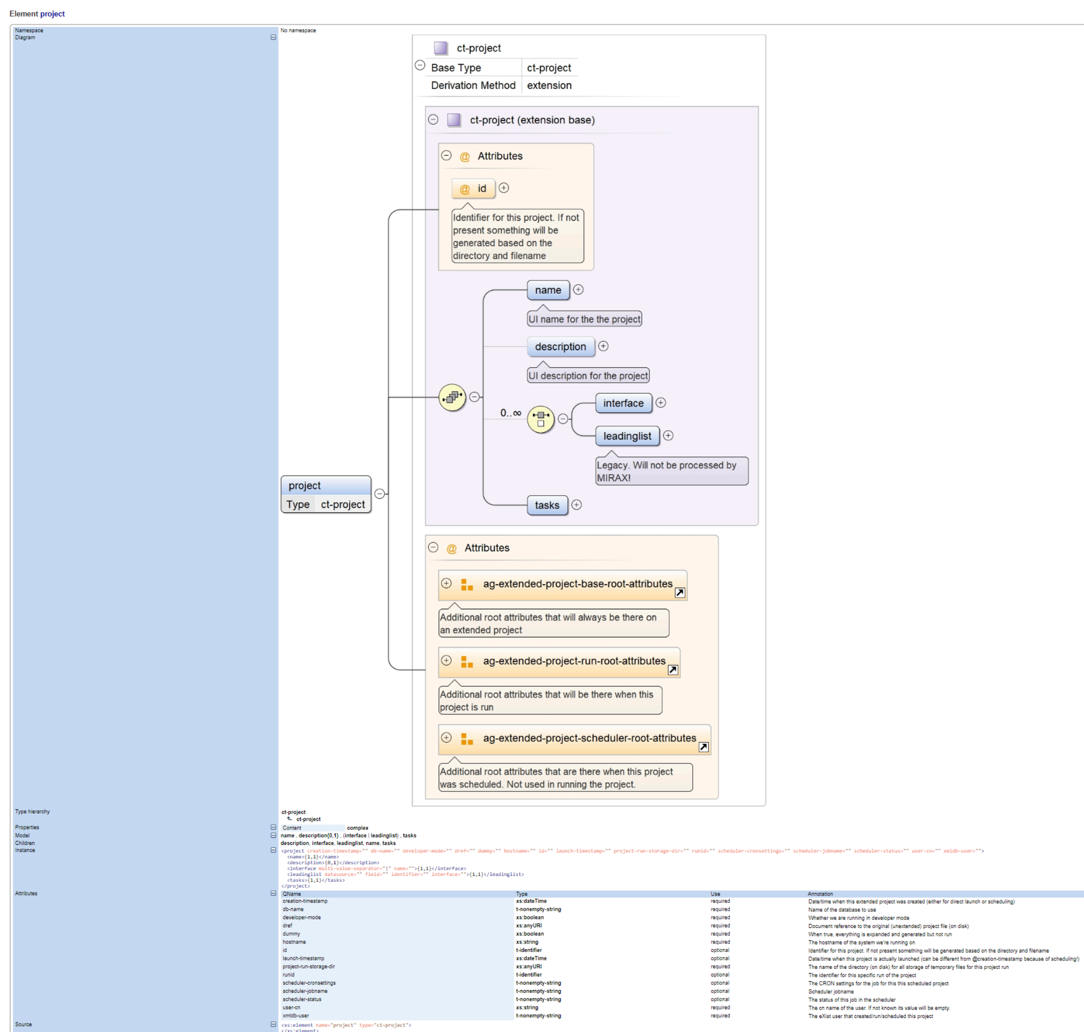
So how to approach this? Before we go in to this, let's make a little side step and look at something that might be going through your mind already: Of course the XML structure in question has a schema of some sort. Can't we use that for (generating) documentation?

3.1. Schemas and documentation

Creating a schema of some sort (e.g. XML Schema or RELAX NG, maybe with additional Schematron rules) for your XML structure is *always* a good idea. It helps users to ascertain they created at least syntactically correct XML. It can be used in IDEs like oXygen to get help in creating the thing. But is it also useable for end-user documentation?

There are generators out there that can turn a schema into documentation. Here is an example of some something generated by oXygen:

Figure 6. An example of (a part of) generated schema documentation by oXygen



Like software has classes, methods and type definitions, schemas have constructs like groups, types, extensions, etc. This is all there for the schema developer and maintainer: keep things consistent, don't define constructs more than once, add internal documentation by using meaningful names and make things easier to change. So a simple element might be constructed from an extended type, adding attribute groups, re-defining constructs made earlier, etc.

All this is very important and necessary, but *not for the end-user*. When you look for instance at the diagram oXygen generates, I think you can see the problem: *way too much detail*. As an end-user of the XML format you're not interested in how the structure is defined in the schema. You simply want to know the attributes and child elements, what they mean and how and when to use them.

To be fair, the oXygen documentation generator can be tuned in excruciating detail and maybe there is a setting I haven't found yet that will generate what we need. But there are two other problems lurking in the wings:

- Are the annotations you write in the schema for the end-user or for the schema maintainer? You probably need both. Maybe using nifty tricks you can keep them apart, but can the documentation generator handle this? You'll also have to be very consistent and careful in creating them.
- XML structure documentation more often than not needs additional narrative texts in-between things. For instance when introducing an element, provide an example or add a warning about something. This does not follow from the formal XML structure, it follows from the flow of the explanation. At this moment there is no way a schema documentation generator can cater for this.

This all is a fundamental problem: Schema documentation generators document the *schema*, not necessarily the resulting format. Lots of unnecessary schema innards show up and obfuscate what an end-user needs to know. You can't add additional texts. Its a bit like trying to generate Java program end-user documentation using the Javadoc pages...

3.2. Writing documentation

So, OK, The idea that you have to write some documentation has landed and you set yourself to this inevitable task. Where to begin? Let's see if we can prepare some guidelines.

- If you're smart, you realize upfront that maybe, one day, you'll have to write documentation of some sort for the structure you're working on. And that means trying try to keep its complexity in check. Especially child constructions with lots of complicated nested choices, sequences, etc. are very hard to explain in a satisfactory way. KISS rules.

- Please realize you're in a knowledge bubble. You know everything there is to know about the XML structure. Maybe you consider it trivial. But your reader does not. So, first and foremost, try to put yourself in the user's mind. What does he or she need to know to use things effectively? What is important and what are details? What does the reader *not* know?

  To reach such a state of mind, the easiest you can do is step away from your project for a day or longer. Take it over the weekend, distract yourself a little. And then plunge in.

- Once you've, at least partially, left your knowledge bubble, try to get into the one of the reader. Who is he or she? What background knowledge can we expect? What is your audience?

  Since we're talking here about documenting XML structures, I think its safe to assume you're writing for a technical, knowledgeable audience. So don't overdo it and ramble about the syntax and semantics of XML in general, or what a schema is, a web-server or other generic concepts. Best to assume this is common knowledge. Writing about it will only annoy your audience and make them feel belittled.

- An often made mistake is to start explaining something without establishing what is *for*. So you happily set out describing your structure's root element and all its attributes, but forget to tell the reader why he/she should bother anyway. Why is this XML structure there? What does it do in general? Where in the processing is it used?

  So it's important to start with some *narrative*. A few paragraphs that explain background, goal, usage and the likes in broad terms. Maybe even some diagram or flowchart?

- Another thing I always like to see before plunging into the details is some *example* (or more than one) of the structure we're talking about. What does it look like? How much of it do I already roughly understand? What's the style? It's like a starter when dining out or the trailer of a film.

  But watch out: deciding on a good example is not easy at all. It must be more than trivial but not overly complicated. It should not bury the reader in frightening details. It should be geared towards some common use-case, one that many readers will recognize and understand and that is not too hard to explain. It must be illustrative.

- Before you start explaining the nitty-gritty details of the XML structure, decide on a *format*. It's not only irritating for the reader to find this element explained this way and another one very different, it will also make your writer's task miserable. This is analogues to programming without conventions. Just try to imagine how tiring it would be to have to come up with some original naming format every time you create a variable, function or class. Much easier to decide on something and stick to it.

  Of course this leads to the question: what is a good, or at least sufficient, format? More about this later.

- Since we're all XML-heads here, I don't think I have to explain the difference between markup and its presentation. So when you're documenting some non-trivial structure, invest time in setting up a tool-chain in which you can describe structures in a formal way that is subsequently converted into the nice diagrams and tables you want to show your readers. Don't create them by hand.

  This is very important for maintenance reasons. Assume for instance you've decide to make extensive and beautiful tree diagrams of your XML structures using some manual drawing tool. The result is awesome and the likes start streaming in. But XML structures, like all software, tend to change over time. So you have to redo some of the drawings, which is a lot of work even for small changes. Nah, not now... The documentation starts lagging behind and probably dies in beauty one day.

- As strange as it may sound, when things get complicated, don't let slavishly following the format get in the way of being understandable. There are situations where your diagrams or formal descriptions will become hard to read and slight deviations from the format you've decided on will improve understandability. Two examples:

  - The XProc programming language has a `<p:choose>` instruction. The (new 3.0) standard depicts it like this:

Figure 7. The **<p:choose>** in the standard

```
<p:choose
  name? = NCName>
    (p:with-input?,
     ((p:when+,
       p:otherwise?) |
      (p:when*,
       p:otherwise)))
</p:choose>
```

Look at the construction for the child elements `<p:when>` and `<p:otherwise>`. It explains, formal and correct, that there must be at least a `<p:when>` or a `<p:otherwise>`. Let there be no doubt: standard's documents *must* be formal and correct, so here it's no problem. But what if you're documenting this? This whole construction with its parenthesis, +, ? and | marks is not really easy to understand for humans. It looks daunting, even while its meaning is quite simple.

So when I was writing the section about `<p:choose>` for the XProc book, I decided to do it a little different:

Figure 8. The **<p:choose>** in the XProc book

```
<p:choose name? = xs:NCName >
  <p:with-input>?
  <p:when>*
  <p:otherwise>?
</p:choose>
```

Somewhat further down followed by the remark:

- A `<p:choose>` must contain *at least* one `<p:when>` or a `<p:otherwise>`. In other words: a `<p:choose>` without at least a single branch is not allowed.

So a simplified diagram, followed by some additional text explaining the border condition. Not formal, not technically exactly correct, but easier and quicker to understand.

These kinds of decisions are also driven by the *importance* of what you're trying to tell. That a `<p:choose>` has `<p:when>` and `<p:otherwise>` children, yes, that's important. That there's a formal condition on there

being at least one... duh. Who will ever want to write a `<p:choose>` without one? You loose a little exactness but you gain a lot of understandability.

- Sometimes possible attribute values are a list of discrete values, like for the `visibility` attribute in following example.

Figure 9. An attribute with a discrete value list

```
<p:declare-step name? = xs:NCName
                type? = xs:QName
                psvi-required? = xs:boolean
                xpath-version? = xs:decimal
                exclude-inline-prefixes? = xs:string
                version? = xs:decimal
                visibility? = "public" | "private" >
```

That's OK when this list is small, but what if it becomes longer and overflows the line width? Since you've decided on a format where discrete value lists are part of the diagram, you might be tempted to put them all there. However, that would mess up the layout of the diagram *and* would forfeit its purpose: a quick, single glance, overview of an element's structure.

A better solution here would be to list just a few values in the diagram, add an ellipsis (...) to signify this list is incomplete and list them (in a table) later on in the document.

3.3. The target format and how to produce it

What should be the target format of our documentation. Can we get away with comments in the file? Will it become a website with linked information? A PDF? Is a wiki page sufficient? Or are we writing a book?

- For files where there's only one of, like configuration files, its tempting to add documentation using comments in the file itself (like the eXist `conf.xml` example above). However, I don't think that works very well. Files are edited, and with the editing, sections, including the documenting comments, will disappear or get duplicated. Your nicely formatted file will soon become a mess. Another reason for not using XML comments is that your layout options are limited. Maintenance of nicely indented and formatted lists or tables in comments is hard to keep consistent...
- When things are easy and extensive documentation is not really necessary (or there is no time), just use what's at hand. Write it in Word and output it as PDF or HTML, use HTML directly, a Markdown readme file, a wiki page, etc.
- When things get a little complicated, more than just a day or so of work and non-trivial, invest in a tool-chain that generates the documentation for you out of some medium-neutral source format. DocBook and DITA are good candidates, especially since IDEs like oXygen have such a splendid support for them. And once you have a medium-neutral source, the final format does not matter much anymore.

  In designing such a tool-chain, begin with the narrative! So start by being able to create "just" text (like a DocBook or DITA structure) and pull in generated diagrams, tables an the likes from there. Not the other way around.

3.4. XML element documentation

What is a good (or at least sufficient) documentation format for our XML structures? We've already talked about the importance of introductory text and examples so let's not talk about that again. Let's look at how to document the basic building blocks of XML: elements. I'll list some requirements first an then talk about what I made of them.

- The structure must be clear in a single glance, with as little ambiguity as possible.
- It must contain all the necessary detailed information, quick and easy to find.
- It must be attractive to look at. Of course attractiveness is subjective but probably everybody knows examples of pages with a really bad layout. Avoid this.
- It must have ample space (and locations) for additional narrative texts.
- It should not deviate too much from what the reader is expecting and used to.

- It must be consistent: all elements must be done the same way (so the reader gets used to the format).

When I started out writing the XProc book I thought long and hard about how to do this. What I came up with is the following:

- For the "clear in a single glance" experience you have to start with some kind of formalized depiction of the XML element. A common way is the following (this comes from the XProc 3.0 standard):

  Figure 10. Example of an XML element documentation diagram

```
<p:with-option
  name = EQName
  as? = XPathSequenceType
  select = XPathExpression
  collection? = { boolean }
  href? = { anyURI }
  pipe? = string
  exclude-inline-prefixes? = ExcludeInlinePrefixes>
    ((p:empty |
       (p:document |
        p:pipe |
        p:inline)*) |
     anyElement*)
</p:with-option>
```

  If you look at other documentation formats, you'll see this in several variations. Admittedly, its hard to beat. It make the element's structure clear in a single glance and people are used to it. So I decided to stick with it, more or less. One important thing I changed is to put angle brackets around child elements to make it very and unambiguously clear that these are *elements*.

  Figure 11. My XML element documentation diagram

```
<p:with-option name = xs:QName
               as? = XPathType
               select = XPathExpr
               collection? = { xs:boolean }
               href? = { xs:anyURI }
               pipe? = xs:string
               exclude-inline-prefixes? = ExcludePrefixes >
  ( <p:empty> |
    <p:document> |
    <p:pipe> |
    <p:inline> )*
  <!-- Or use any other element(s) (or plain text) as alternative for
       wrapping it in a <p:inline> -->
</p:with-option>
```

  Regardless of the layout details, starting with something like the above is important. People are used to it and get an immediate and clear impression what we're talking about.

- After this diagram come the details. What do the attributes and child elements *mean*? How can we make this as clear and easy to find as possible?

In my view, nothing beats tables in presenting this kind of information. These tables should be consistent in that *all* attributes and child elements must be there. Yes, even when you're documenting this `name` attribute for the umpteenth time, it *must* be there. The reader might not have seen the explanations that came before and *expect* it to be there (but nothing of course that stops you from pulling repeated texts from some common source in your tool-chain...).

Here is an excerpt of the table explaining the attributes of the example `<p:with-option>` element:

Figure 12. Example of a table explaining attributes

Attribute	#	Type	Description
name	1	xs:QName	The name of the option.
as	?	XPathType	The data type of the option.
select	1	XPathExpr	An XPath expression that determines the value of the variable.
collection	?	{ xs:boolean }	Default: false If true the XPath context item for setting this variable will be undefined. All documents are now available as the *default collection* (through the collection() function. See "Addressing multiple documents" on page 108.

And here is the table explaining the child elements:

Figure 13. Example of a table explaining child elements

Child element	#	Description
p:empty	*	Use the empty sequence (a.k.a. nothing) as the input for this option. See "Specifying nothing: p:empty" on page 110.
p:document	*	Use a document read from an URI as the input for this option. See "Reading external documents: p:document" on page 111.
p:pipe	*	Use a connection to an output port of another step (or the input port of the step itself) as the input for this option. See "Explicitly connecting to another port: p:pipe" on page 112.
p:inline	*	Use the children of this element as the input for this option. See "Specifying documents inline: p:inline" on page 113. If you only have a single `<p:inline>` child element, you can leave it out and state your inline documents as direct children of `<p:variable>`.

As you can see, the texts in the tables are relatively short. When longer or additional explanations are necessary, put them elsewhere and link to them (or say something like "see below").

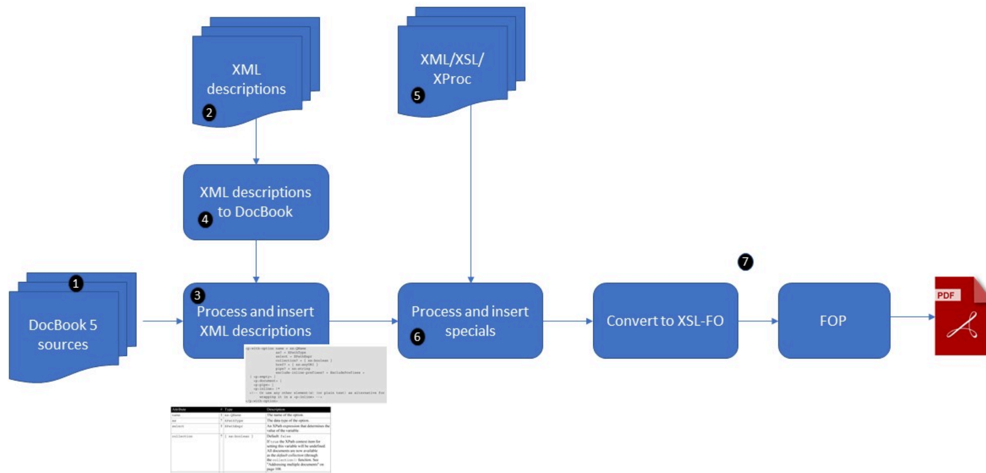
- I will not show you, but my tool-chain allows me to add additional narrative texts (almost) everywhere: in-between the diagram and the tables, in-between the tables, etc. So whenever and where-ever necessary, additional explanations can be added, exactly where they're needed to have maximum impact on the understandability.

So is this it? No, absolutely not. I regularly catch myself changing this or that little detail (especially when the inspiration for writing prose is temporarily missing, the equivalent of doodling around...). There are still some things I'm not totally satisfied with but unsure on how to proceed or decide. For instance the order of attributes/child elements. Should this be alphabetical or in order of importance? And if the latter, how do you decide on importance? It's probably something that is never finished but one thing I do know: it's slowly getting better.

3.5. An example tool-chain

As an example a short overview of the tool-chain I'm using for creating the XProc book. And guess what: it's written in XProc, although of course still V1.0.

Figure 14. My tool-chain for the XProc book



1. The tool-chain starts off with DocBook 5 sources, split into files per chapter.
2. The descriptions of the various elements are in separate XML files, using a (self invented and, guilty as charged, badly documented) XML dialect.
3. Inside the source DocBook there are special instructions (XML elements in a different namespace) that trigger the processing of these XML descriptions
4. A special sub-pipeline converts the XML descriptions into DocBook and takes care of all the formatting, table building, etc.
5. I don't always want to create things like complex tables directly in DocBook. That's hard to write and maintain. Instead I write some XML that contains the data and a conversion (either XSLT or XProc) that turns it into DocBook.
6. Like the XML descriptions, this is triggered by special instructions in the DocBook sources.
7. Finally the tool-chain converts the DocBook into XSL-FO and, through FOP, into PDF.

All the components of this tool-chain are in an open-source library on GitHub (https://github.com/eriksiegel/xtpxlib, in the `xdocbook` folder). There is a little documentation. If you're interested and need help, drop me a mail and I'll see what I can do.

4. Conclusions and wrap-up

As a wrap-up, let's try to summarize a few things:

- All this is relevant when you have invented some XML structure that will be used by others and you want to make sure they'll understand it.
- When you're serious about documenting, invest in a tool-chain that automates at least the generation of element/attribute documentation.
- Acknowledge you're in a knowledge bubble. Try to distance yourself from what you know before you begin.
- Always add a schema of some sort. But that's not enough.

- Take the narrative as a starting point and include generated parts *into* your (hand-crafted) work. Generated documentation is almost never sufficient, satisfactory and/or clear enough.
- Start by taking the reader by the hand in some overview introduction. Provide one or more examples.
- Spend some time making it look *nice*.

XMLPaper: XML-based Conference Paper Workflow

Cristian Talau, Syncro Soft SRL

Abstract

Popular conference management systems are monolithic solutions that are used across many fields and by very diverse audiences. As a result, these systems have rigid technology choices for paper format: PDF, TeX or Word which have limitations compared to a structured format such as XML. In addition, they do not cover all the steps in the paper submission lifecycle such as: collaboration between authors.

We propose a new solution for conference paper submission management that tries to improve the user experience in several areas: authors tooling setup, collaboration between authors, multi-step review processes, multi-format proceedings publishing. This paper will present a typical conference paper submission workflow and identify its steps, the stakeholders and the tools and technologies used. We will then present how our solution improves user experience of each of the stakeholders.

In this solution, the documents are authored in the an XML format, that supports publishing both as PDF and web portal. The output format is customizable and clearly communicated to authors so that they can preview how the paper will be published. Authors have a very intuitive user interface to draft the paper and to collaborate in cases where a paper has multiple authors. Reviewers can choose to review either the published PDF or directly on the source. In case of multi-round reviews, they can see the changes made between different revisions, thus being able to focus on the last updates.

From a technical point of view, the solution is composed from off-the-shelf Web services with a thin layer of orchestration. It may to be used together with a regular Conference Management System, replacing parts of its functionality.

To conclude the paper, we analyze how this solution is similar with other workflows related to content creation in a company, such as creating release notes or datasheets for a product.

1. Introduction

In this paper I will first present the lifecycle of a conference paper, the key actors and the document formats used. In the next section I will present the architecture of XMLPaper. Then I will explain how this solution improves the user experience for each of the stakeholders.

To conclude the paper I will point out similarities between conference paper submissions management and other content management workflows that usually happen in a company.

2. Conference paper submission workflow

2.1. Content creation

A conference paper submission lifecycle starts with the authors collaborating to create the content of the paper. At this point the may request preliminary reviews from colleagues or other members in their organization.

The most popular format used to submit a paper is TeX for science conferences while Word is the de-facto standard for humanities conferences.

It is not uncommon for authors to create drafts in other formats and only the final submission to be written in the format required by the conference.

2.2. Peer review

After the paper is ready it is submitted to the conference committee to be reviewed. The review can be a multi-step process in which authors iterate on the paper after a round of reviews.

This review is usually performed on the PDF or Word version of the paper and comments are written in the Conference Management System and not directly linked to parts of the document. For multi-round reviews have to identify the changed parts of the paper and to evaluate whether their previous feedback how addressed.

2.3. Publishing

After the paper is accepted, an editor needs to put it together with the rest of the accepted papers and edit them to have a consistent style and format. At the end the proceedings of the conference are published.

Most of the conferences publish only a PDF version of the proceedings which is also used for print in some cases.

To make sure all papers have a consistent format the conference employs either a Word styles library and expect authors to adhere to those. In other cases the conference publishes a LaTeX `.class` file to be used by all authors and expect that the submitted PDF files were generated using that configuration file.

3. XMLPaper architecture

3.1. The source format

XMLPaper uses the DocBook format for the paper across all the stages of its lifecycle.

DocBook has special markup for articles and is rich enough to support embedded objects like images, videos, mathematical equations, tables, code blocks and so on. However, DocBook is not the only suitable XML vocabulary - JATS (Journal Article Tag Suite) is also a viable alternative.

Some advantages of using DocBook include:

- The authors can focus on creating the content of the paper and leaving it to conference organizers to define the layout and style of the published version of the paper.
- The content can be published to multiple formats, such as PDF, web page and e-book. Having the source available, one can re-publish it later as new formats become popular, e.g. audio paper.

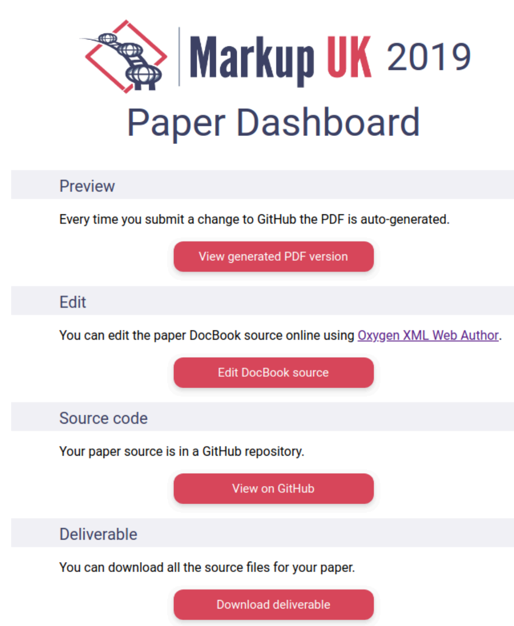
- Being an open standard format, it is good for archiving purposes. Some academic papers have a longer lifetime that the company than a particular proprietary format.
- Being widely used in enterprise content creation, there is wide range of tooling available for XML-related tasks such as editing, collaboration, and publishing.

3.2. Self-service project template

The delivery format of XMLPaper is a self-service project template. This project template does not only contain an outline of a paper to be filled in, but also configuration files for all the required tools and services so that authors can start working on the conference paper right away.

In order to create an affordable solution we reused off-the-shelf tools and services and made them work together without having the author set every up thing. To make it easy to create a portable configuration that works without any software requirements for authors XMLPaper uses mostly cloud services.

XMLPaper allows authors, just by clicking a button, to copy the project template to their Git account and deploy a web-based dashboard for working on the paper.



XMLPaper Dashboard

The Dashboard contains buttons to perform the following tasks:

- Preview the latest committed version of the paper in PDF format.
- A link that can be used to edit the paper in Oxygen XML Web Author.

  The project includes conference-specific editor configuration:

  - It validates the paper source "as-you-type" according to the conference-specific constraints expressed as Schematron rules.

    Editor issues a warning that the abstract is too short

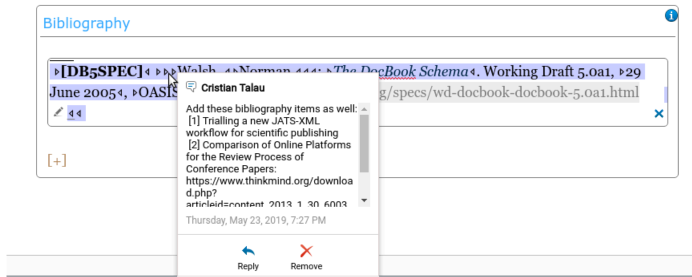
  - It includes inline actions and hints to help authors create the correct XML structure

You can organize your article in multiple sections and sub-sections. It is recommended to start with an "Introduction" section and end with a "Conclusion" section followed by bibliographic references.
There are a few actions available that allow you to add a new top level section or to promote or demote a section like changing between level 1 and level 2 sections, for example.

Section 1: Introduction

Inline hint about how to split the paper in sections. Inline actions to convert a section to subsection and vice versa.

- The change tracking and commenting features help co-authors collaborate. They can also be used to send document for review before submission.



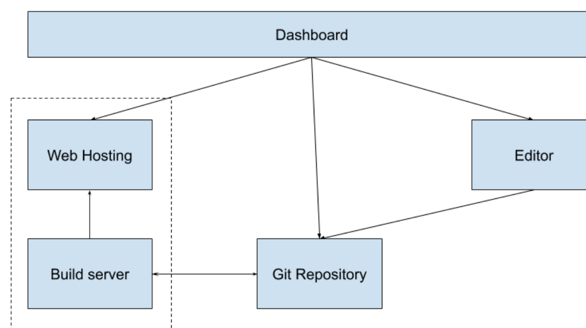
Comment added by a reviewer suggesting some additional bibliography items

- View the XML source code in the Git repository. Most repository hosting services can be used for collaboration between co-authors - they offer version tracking, issue management, and other collaboration features.

  In this repository, users can find an `.xpr` project file for Oxygen XML Editor which configures the editor to match the conference requirements similar with the web editor. It also includes a Transformation Scenario that generates a PDF preview. Using this editor authors can edit and generate a PDF preview of their paper while offline.

- Download the paper in XML format and all supporting resources (`xi:include`d files, images, etc.)

The back-end of the solution is composed of several services as depicted in the diagram below:



XMLPaper back-end services diagram

- The Dashboard is a static web page with hyperlinks to the web interfaces of the other services. It is generated during the initial setup since the links depend on the location of the Git repository.
- The Git repository stores the source of the paper together with configuration files and build scripts.
- The web-based editor can be opened using a link on the Dashboard. It connects to the Git repository to read the paper source and to commit it every time the users saves.
- Every time an user commits a file in the Git repository, a git hook triggers a new job on the build server. This job connects back to the Git repository to pull the entire repository and then performs the following tasks:
  - Generates the PDF preview.

This task involves downloading several tools such as Saxon 6.5, DocBook XSL, XSLT-HL (for syntax highlight in code-blocks), and Apache FOP. It also involves applying some custom XSLT and configuring the above-mentioned tools.

Example 1. PDF Generation configuration

```
java -cp "bin/saxon.jar:bin/docbook/xsl/extensions/xslthl.jar" \
    com.icl.saxon.StyleSheet processed_paper.xml \
    bin/docbook/xsl/fo/docbook_custom.xsl \
  admon.graphics=1 \
  admon.graphics.extension=.png \
  admon.graphics.path=bin/docbook/css/img/ \
  "body.font.family=Times New Roman, Tahoma, Batang, serif" \
  callout.graphics.path=bin/docbook/xsl/images/callouts/ \
  draft.mode="no" \
  fop.extensions=0 \
  fop1.extensions=0 \
  highlight.source=1 \
  highlight.xslthl.config=\
    file://`pwd`/bin/docbook/xsl/hl/xslthl-config.xml \
  "monospace.font.family=monospace, Courier New, Consolas, Arial" \
  paper.type=A4 \
  "title.font.family=Arial, Tahoma, Batang, sans-serif" \
> out/paper.fo
```

However, configuring the publishing pipeline is the job of the conference organizers. Authors just save in the editor and then a build starts which in turn uploads the generated PDF to the web server.

- Generates the deliverable archive containing the XML source and all its referenced files, for example images.
- Generates the Dashboard web page. This page is static and does not change over time. However, its links depend on the URL of the Git repository, so it has to be generated during the first build.
- Pushes everything on the web hosting service. In the current implementation the web hosting and build server services are part of the same product.

3.3. Steady-state workflow

The steady state workflow is simple and it resembles the one used for static-site generators:

1. User saves their changes
2. The editor commits the changes in the Git repository
3. A git hook is called to trigger a job on the build server
4. PDF and other reports are generated
5. Artifacts are uploaded to the Web Server and ready to be used.

3.4. Initial setup

The initial setup is more complicated than duplicating the template project since it needs to link together multiple services offered by different vendors. For more details about service choices one can consult the web page of the XMLPaper project: https://github.com/oxygenxml/markupuk-2019-paper.

To make the setup more intuitive XMLPaper uses a setup tool that takes care of connecting various services. This setup tool is part of the same suite as the build service and web hosting service. To make other services cooperate, the steps below are performed:

Authorize the setup tool to create a Git repository
: The setup tool uses OAuth to ask the authors for permission to create and manage a Git repository in their account starting from the project template.

Link the build service to the Git repository
: The setup tool registers a Git hook to trigger the build service whenever a commit is pushed. It also registers an SSH key so that it can fetch the repository

	contents to perform the build. Each build job will have the URL of the Git repository as an environment variable.
Generate the Dashboard web page with links to the other services	The build service runs a script found in the project that uses the URL of the Git repository to create back links to open the source of the paper either directly or in the web editor.
Authorize the web editor to access the Git repository	When the web editor is opened for the first time, it uses OAuth to request the user's permission to read and commit files in the Git repository when the author opens or saves the paper in the editor.

Note

Some of the services require the author to create an user account. In our case, the user has to create an account with a Git repository hosting service and another account for the build service, setup tool, and web hosting service. The user account creation usually happens as an interstitial step and only if the user does not already have an account.

In our case, the services are either part of the same bundle like the build service and web hosting service or are connected using OAuth. Other services use personal access tokens to allow applications access their API. In this case, the user should be guided to generate one and give it to the setup tool.

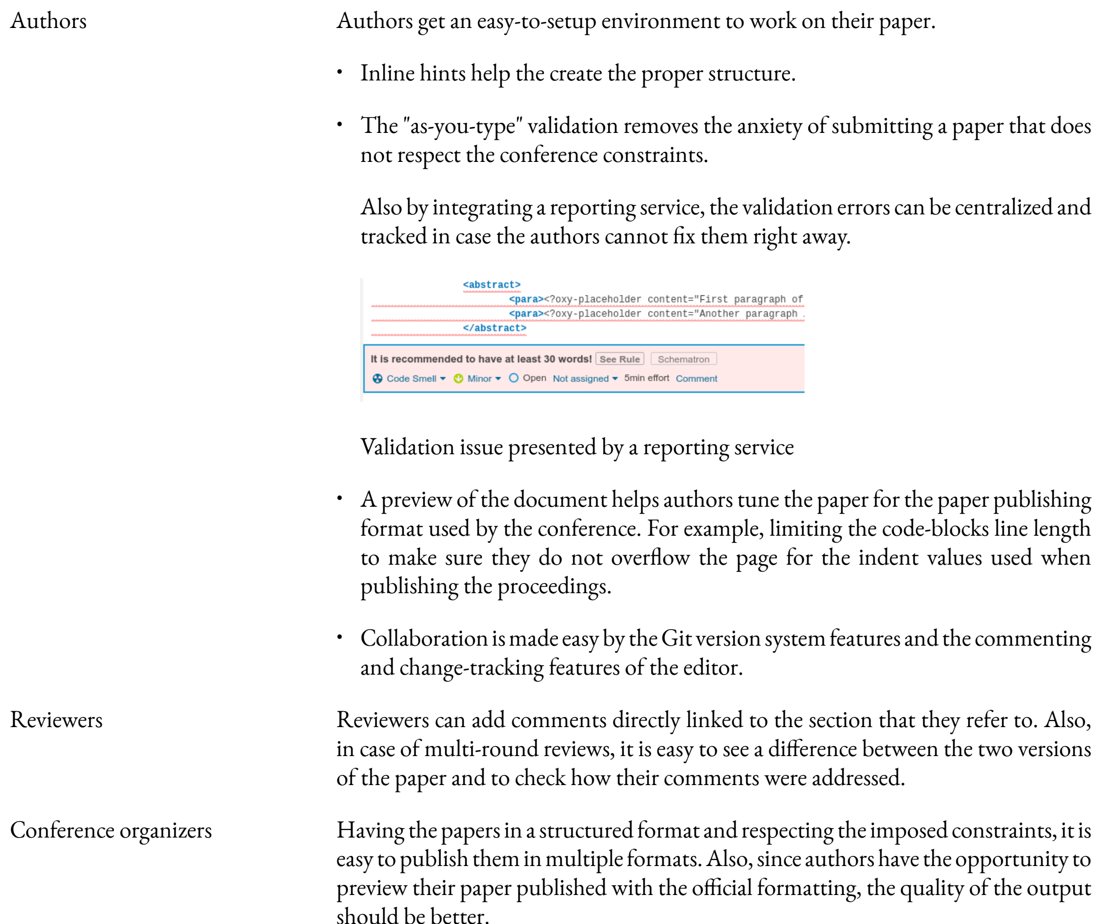
4. User experience analysis

Authors

Authors get an easy-to-setup environment to work on their paper.

- Inline hints help the create the proper structure.
- The "as-you-type" validation removes the anxiety of submitting a paper that does not respect the conference constraints.

  Also by integrating a reporting service, the validation errors can be centralized and tracked in case the authors cannot fix them right away.

  Validation issue presented by a reporting service

- A preview of the document helps authors tune the paper for the paper publishing format used by the conference. For example, limiting the code-blocks line length to make sure they do not overflow the page for the indent values used when publishing the proceedings.
- Collaboration is made easy by the Git version system features and the commenting and change-tracking features of the editor.

Reviewers

Reviewers can add comments directly linked to the section that they refer to. Also, in case of multi-round reviews, it is easy to see a difference between the two versions of the paper and to check how their comments were addressed.

Conference organizers

Having the papers in a structured format and respecting the imposed constraints, it is easy to publish them in multiple formats. Also, since authors have the opportunity to preview their paper published with the official formatting, the quality of the output should be better.

5. Similarities with other content creation workflows

The demand for more XML content has shifted the work of creating it from technical writers to other departments of a company. Since these people work only occasionally on content creation the investment to have them set up the necessary tools does not pay off.

The idea of having a self-service project template that has all the configuration in place can be applied in other scenarios such as creating similar documentation for multiple products. For example, customizable products whose documentation has a standard format but the parameters can differ, datasheets for microchips or release notes for a software product.

For cloud-based services, the ease of setup is at odds with other business driving factors such as security, legal compliance, and brand exposure. For example, if a company provides a back-end service as using a freemium pricing model, they want the user to be aware that about that service. So, it is not uncommon that during the setup users will be exposed to a screen presenting that service.

However, in an enterprise environment, these factors are not that relevant which can provide an even more intuitive user on-boarding experience. For example, a back-end service can is usually licensed by the company and configured by a system administrator and is totally transparent to the end-user.

6. Conclusion

The paper presented how a serverless solution for an XML-based paper submission workflow can be developed with relatively low effort.

An important observation is that although the solution is built from a number of off-the-shelf tools and services, most of them are not specifically designed for XML-based solutions, but for web applications.

The source code for the project is available online at https://github.com/oxygenxml/markupuk-2019-paper. We plan to customize it further with conference specific constraints expressed as Schematron rules. We also plan to make it possible to generate the PDF preview on demand without having to commit the paper in the Git repository first. This will imply integrating with a Functions as a Service platform that will run the publishing process.

Bibliography

[JATS] Hassan Tamir: Trialling a new JATS-XML workflow for scientific publishing. 16 February 2019, Round-Trip PDF Solutions http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf

[PLATF] Lorena Parra: Comparison of Online Platforms for the Review Process of Conference Papers. 2013, Universidad Politécnica de Valencia https://www.thinkmind.org/download.php?articleid=content_2013_1_30_60033

[REV] Mark Bernstein: Reviewing Conference Papers. 2008, Eastgate Systems Inc. https://www.markbernstein.org/elements/Reviewing.pdf

[OAUTH] RFC 6749: The OAuth 2.0 Authorization Framework, IETF https://tools.ietf.org/html/rfc6749

[PROGIT] Scott Cachon: Pro Git. 2014, https://git-scm.com/book/en/v2

Dispelling Myths About Markup Formats: When What Why Where

Liam Quin, Delightful Computing

Abstract

Misunderstandings about the goals and strengths of different document and data interchange formats can lead to suboptimal decisions. Such misunderstandings appear widespread. The purpose of this paper is to suggest areas in which each format has strengths, and to provide clear explanations that people can use to place XML in the context of other current markup systems.

Common misconceptions about XML include statements such as "XML was designed for Web services and therefore unsuitable for documents;" "XML was designed to replace HTML and has failed;" "XML cannot transmit semantics of any kind;" "XML is dead." In fact, XML is alive and well. There are misconceptions about other formats, too, of course.

1. Introduction

Declarative markup languages have been around for many years. There were declarative systems built from Unix *troff* macro libraries in the 1970s and 1980s, and of course SGML, Scribe, LaTeX and many others attempt to be more or less declarative and more or less general ways to represent and process documents with computers.

The choice of which markup system to use should be based on clear well-understood criteria. These might include:

- Which notation has a closest fit to the document or documents. For example, a document with a lot of mixed markup and running text will be more difficult to process in some systems than others, as will a multilingual document;
- Which notation has associated tools that perform the processing the project needs;
- Which system the developers like the most or hate the most;
- Which system will be most likely to have active support for the entire duration of the project;
- What will be the costs and what will be the gains of each format;
- What are all the other similar projects using.

Very often, the emotional aspects of the decision outweigh the technical aspects. This paper primarily explores the technical aspects, but also provides the reader with some ammunition to make an effective presentation in the emotional arena.

The examples of criteria enumerated above do not include the single most important aspects of a decision: the context and the situation. The *context* is an emergent property of organizational culture, fashion, predicted usage, and much more; some of this is listed explicitly here and some not. The *situation*, or the circumstances around the decision, must be viewed in terms of the wider context: instructions for identifying radioactive material in a nuclear power plant might have to be understood for a hundred years or more; a financial transaction must be archived for a period determined by a legal statute of limitation; British laws are written on vellum and stored in a stone tower in some cases for a thousand years or more. But over the lifetime of the information the context in which it is used may change: a financial record might be read by corporate accounting staff or, later, by a government auditor.

The available tools will vary, but the organization may have a legal need to make information available regardless of applications originally used to process it. Similarly, software applications, operating systems, even display and user interface hardware constraints, all change over time. But a bigger question is who determines the format: who determines what information is stored, and how, and how is it to be used. A document that is read by an application to configure user preferences is likely designed by a programmer; a transcription of a mediaeval manuscript is more likely to be represented in ways determined by the people working with the text itself rather than with any particular application.

The following sections attempt to identify good and less successful contexts for different markup formats, and, again, to provide ways of describing these contexts.

1.1. HTML: The HyperText Markup Language (HTML).

The first standardized version of HTML was described by a combination of prose and an SGML DTD. In this sense, and one other (see XHTML later in this paper) HTML shares some ancestry with XML However, people working with XML and with HTML often have very different ways of looking at markup. The difference can be illustrated by the meaning of the word “semantics” in the respective communities. In the HTML world, it’s common to hear people to describe the meaning of an element in terms of what a Web browser does when it encounters the start and end tags. Although most browser developers no longer refer to start and end tags as separate commands, the idea remains that operational semantics, Web browser behaviour, is the primary focus of HTML design. Marking up in the sense of identifying the content of a document seems alien here: there’s no *poem* element, for example, and tie *cite* element has no standard way to identify the author of a quotation or to give a bibliographic reference.

HTML offers some extensibility: one can use markup such as:

```
<span class="volno"3</span>
```

to indicate the issue number of a volume containing a journal within a bibliographic reference. As with “plain XML” there is no standard way to mark up a bibliography and one is actually more likely in practice to find simply

```
<b>3</b>
```

in practice, or, worse,

```
<span class="cn506dw"3</span>
```

inserted by a content management system, framework, or word processor conversion.

IHTML 5 includes elements such as *nav* and *main*, but their goal is to help people and programs navigate round documents by identifying the functions of different parts, not to try and label the parts and have the browser automatically function appropriately.

HTML, then, is strongest when the Web is the primary end output format. It should be mentioned that there are also products that support generating PDF for print from HTML and CSS, and that the EPUB 3 standard uses XHTML (with a move to HTML possible in the future).

A difficulty with HTML can be that the content can be difficult to reuse or re-purpose. If you do the work to annotate your HTML so that audiobooks can be made, or to use *class* attributes consistently enough that your aircraft repair manual can be generated automatically from your twenty-thousand-page operations manual, you will be replicating the work that might already have been done for you with an SGML or XML system: the tools are not generally designed for large, long, complex documents with precise domain-specific requirements. The primary application domain and usage context of HTML is the Webb browser.

A *benefit* of using HTML, however, can be reduced staff training. If your writers are already familiar with HTML, they can be up and running quickly. Beware, however, that saving a few hours of training does not end up costing you weeks or months of work when you discover the files are not consistently marked up or contain errors that weren't flagged by (error-tolerant) HTML systems or Web browsers.

2. Markdown

The name *Markdown* is of course a play on *markup*, with the implication that it's somehow *less*: less work, not to hard to understand. For very simple documents Markdown can indeed be easy. In the manner of DEC NotesFiles from the 1970s and 1980s, one uses ASCII symbols for *bold* and _italic_. Hypertext links are more complex, and tables are supported only in some of the Markdown variants and can be a nightmare to use in a text editor. The primary context for using markup is where people edit the text directly in a plain text editor and this means that it is really only intended, and only suitable for, simple documents. If you are using a tool that edits Markdown and hides the syntax then you might as well be editing HTML or XML: the primary reason to use Markdown files is that you have tools, such as github or a wiki of some sort, that consume it.

Since there are no special tools needed, and since Markdown can really only cope with simple documents, the syntax can be easy to learn and training costs can be low. Markdown is generally close to a subset of the format used by Wikipedia, so, with extensions as needed, it can clearly scale.

The semantics recorded in Markdown is purely operational: go bold, go italic; whilst some variants have additional tagging support, the primary assumption is that the appearance of the document (or the sound, when read out loud by text to speech) is primary.

Markup is *fabulous*, though, for very simple README documents for projects such as computer source code, where the file can be read easily in a plain text viewer or editor without any special tools.

If you choose to use Markdown for a project, be aware there are competing versions and make sure you have a fully compatible tool-set if you need one. If, however, you have figures, tables, footnotes, cross-references, running page headers (or even page numbers), or other features that go outside the normal remit of Markdown, or if you foresee a need to reuse the information and extract information from the text, you will almost certainly be better off with a richer format.

If necessary, you can provide a tool to read a specific version of Markdown and convert it to (say) XML-encoded TEI documents for users in a text-based environment.

3. RDF and Linked Data

The Resource Description Framework (RDF) is really a data model for exchange of knowledge representation. Linked Data conforms to the same model: triplets of (subject, relationship, object), where all three terms are URIs (Uniform Resource Identifiers), so that if two triplets use the same URI for the same subject then the properties from both triples apply to it, and so on.

RDF is *not* a model for documents. One can extract triplets from documents, but if it is to be useful the extraction is normally lossy: a list of the characters and places mentioned in a novel such as *Star Wars* is a common example, together with externally-derived triplets describing relationships such as *was born at* or *loves* or *caused to become frozen*. The list

does not include every occurrence of every word in the novel, and the book cannot be reconstructed from the list. This example is very typical in its relationship to information, to the original book. Query languages such as GraphQL and SPARQL can then be used to explore the data and to support applications. Note that Linked Data may in many cases represent a complete data set, but even there the data is normally unordered, unlike words and paragraphs in writing.

A weakness of RDF is that triplets are not natively labeled as to their source. In practice a *triple store* (a content management system for RDF) will normally extend the model to add permissions and information about who added each triple, but this is not normally accessible to the standard query languages. RDF stores may also support federated queries, in which case the additional information is not passed between systems, potentially leading to trust and security issues.

The RDF model is also used for the Really Simple Syndication format (RSS), so that RSS files are simple XML representations of an RDF data model.

RDF values are simple (atomic) strings: there is no direct support for mixed content such as running text in HTML, and text fragments containing element markup, such as one finds in RSS, have to be escaped and are treated by RDF systems as simple strings. RDF query languages tend to be weak in string processing, exacerbating difficulties here: you can't generally use SPARQL to find all RSS feed items containing an HTML link to a given resource, for example.

Linked Data in all of its forms provides ways to think about property-based reasoning, and can be very useful when working with metadata, especially in conjunction with other document formats.

4. JavaScript Object Notation (JSON)

JSON rapidly became very popular in the realm of Web development. It uses a mixture of *array*, which are ordered sequences of JSON items; *objects*, which are unordered collections of key-value pairs in which the key is a string and the value is any JSON item,;and, finally, values such as integers and strings. This is sufficient to represent static JavaScript objects (there are no function items).

JSON has taken over from XML for the purpose of exchanging program-generated data via distributed APIs (largely replacing the use of SOAP) and for configuration of Web-based software. The notation uses square brackets for arrays and curly braces for objects, making it familiar to programmers of C-like languages and also allowing automatic brace-matching in many widely-used text editors.

Large JSON files can be difficult to edit directly, as you can end up with large numbers of closing braces in a clump, reminiscent of LISP. It's slightly easier than HTML in this regard, though, where you can end up with masses of *</div>* tags together, given that text editors tend not to support tag-matching (show me the start tag corresponding to this close tag) but do support brace-matching.

JSON does *not* support mixed content: that is, you can't have running text with embedded markup. You can *represent* mixed content with an array of mixed strings and objects, but is not suitable for manipulation without tools or programs, and is considerably more verbose than the corresponding Markdown or even XML or HTML markup in this case. It's also easy to lose significant spaces around the embedded markup when working in this fashion.

There are libraries to import JSON files, and often to export them, in all major programming languages today. Often, reading JSON results in language-native objects, so that no query language is needed in simple cases. This is a marked contrast from XML or HTML, where the result is usually a relatively complex data structure.

JSON, then, is strongly favoured by many developers over any of the other formats in this paper because it is much easier for them. JSON belongs in a context in which the format of the data, the exact representation of information, and even the choice of what is to be represented is in the hands of programmers. In contexts in which document authors own their data and choose the contents, or where interoperability of the information between applications is paramount, JSON is much weaker than XML or RDF. However, see the next section for JSON and RDF.

5. JSON-LD: JSON meets RDF

JSON-LD is a way to represent linked data in JSON. It provides a mechanism to reduce the verbosity of the result by a *context* that authors can use to give short names for URIs in the data.

JSON-LD is intended for environments and contexts (in the sense of this paper) in which JSON is already in use and there's a need to add support for linked data using the RDF data model, perhaps to support graph-based query languages.

Although JSON-LD combines the advantages of both JSON and RDF, it also inherits many of their disadvantages: the underlying data model of RDF is not extended to include the source of triplets, and mixed content is still not directly supported.

6. Portable Document Format (PDF)

A disadvantage in some applications of all of the above formats is that they are not easily printed in a useful way. HTML is the closest, as Cascading Style Sheets (CSS) can be used with formatting software to make print-ready pages, but it is necessary to write separate CSS stylesheets for different HTML documents. The result of formatting HTML for print is usually PDF, which raises the question of whether PDF is useful as a document interchange format.

PDF documents can be printed: PDF is a *page description language* and a PDF file describes the exact location of the page of everything inside it, both text and graphics. Unfortunately for document interchange this also means that PDF files cannot be edited to insert or delete content: the remainder of text on the page does not normally reflow.

The PDF format contains machinery for embedding fonts, and for complex graphics. It is a compressed form of the PostScript language. Using PDF can provide *page fidelity* in many cases, in the sense that printed pages will look the same everywhere, scaled if desired to print on differently-sized paper.

Since PDF is not easily edited, and has limited accessibility to people who are blind or who cannot distinguish certain colours or who need low (or high) contrast, it is not suitable as a primary document format.

Reading and processing PDF in programs is difficult; there *are* libraries for C and JavaScript and other languages, but the resulting data structures are complex. There are no standard query languages for PDF, and it should be noted that even determining natural-language word-breaks is unreliable, using heuristics based on the position of each letter of the word on the page to try and detect the spaces. In addition, it is not usually possible to distinguish between a line-break at a hyphen in the original input and a hyphen inserted to facilitate line-breaking: text formatting is *lossy* so that the original document cannot be reconstructed reliably.

7. Domain-specific XML Vocabularies

Like RDF, XML is really a framework for one's own information: it does not come with much in the way of predefined semantics, whether behavioural or extrinsic. Like HTML, XML has a simple tag-based syntax for representing elements, but unlike HTML, XML does not predefine any element names. There is also no single data model for XML, although a few data models predominate in practice, primarily DOM and XDM.

Since XML does not predefine element names, and also does not support anonymous (unnamed) elements, one needs a set of XML names for elements (and their attributes) to use. A set of element names and constraints on them can loosely be called a *vocabulary*. To make XML useful, there are three common paths people take, depending on their situation and the project context: to use a domain-specific XML vocabulary; to extend an existing XML vocabulary; to develop a custom XML vocabulary. Subsequent sections will describe each of these in turn, including some examples and some exceptions.

Some frequently-heard complaints about XML should be mentioned here. People say that XML files are large compared to, for example, JSON; this is often true although XML compresses well. But the names repeated in XML end tags and the quotes around attribute values also provide a level of redundancy: experience with SGML minimization showed that the ability to omit these increased support costs considerably, because they created common situations in which the document would parse and even validate, but the interpretation of the parser differed from that of the human user.

Another common complaint about XML from developers is that the APIs for working with XML are inelegant. Although this was true fifteen years ago, today there are often much more convenient APIs, ranging from XQuery and XPath to JQuery. But in any case the primary question of context for an XML project is whether the document maintainers are in control of the markup or whether the developers are. The former case is a primary use-case for XML. When developers are in control, JSON may be a better fit for short-form key-value or object-like documents. XML remains the format of choice for mixed content, where there is a mixture of text and markup at the same level.

8. XHTML™: The Extensible HyperText Markup Language

XHTML is a widely-used XML vocabulary that is also a reworking of HTML into XML. There are two primary versions: 1.*x* and 5. Version 5 is the current version, and is an XML syntax for HTML 5. The original XHTML 1 versions supported customization using XML DTDs, the original mechanism to define an XML vocabulary. XHTML 5 does not use grammar-based validation, and is primarily intended for use in Web browsers. XHTML is also used by the EPUB standard for electronic books, where it has the advantage that a device-specific rendering engine can be used without having to worry about full HTML compatibility. This is important because Web browsers tend to be very flexible in displaying files containing errors, so existing content often contains a lot of syntax errors, which in turn means any software that tries to process arbitrary HTML needs a relatively complex parser and a lot of compatibility code. With XHTML, the syntax checking is strict, and enforced by the XML parser.

Since XHTML files can be read by XML parsers, they are amenable to processing with XSLT and XProc, and to being stored in databases and queries with XQuery. Modern versions of XSLT and XQuery can also create XHTML 5 files.

XHTML, like HTML, is designed in the context of Web browsers. Like all software, Web browsers evolve. Over time, elements change meaning or are dropped entirely. However, HTML 5 dropped the idea of including a version number in HTML files, which may cause problems with long-term archiving. Of course, any vocabulary could change over time, but the mitigation there is to combine grammar-based validation and specific version marking; HTML and XHTML do neither.

XHTML is good for projects at the edge of XML and the Web, because they are equally usable by both tool-sets. The element names are also understood by Web search engines, so that serving XHTML files directly on the open Web makes sense and works. XHTML 5 is a good choice for generic documents such as blogs, as well as for Web-based applications. It has deficiencies in validation compared to domain-specific vocabularies, but it would also be fair to consider XHTML to be a domain-specific XML vocabulary for sharing documents on the World Wide Web. The name (X)HTML is sometimes used to refer to the vocabulary independently of whether the XML or slightly different HTML syntax is used.

If you are producing your own domain-specific vocabulary, it is worth considering using (X)HTML 5 element names for plain text paragraphs and for markup within them, simply because (X)HTML is, overall, the most widely-used vocabulary on the planet. However, beware of false promises: if you use *p* for paragraphs, people may expect to be able to use *ol* for a list, *i* for italics, and so on.

9. DocBook

DocBook is a domain-specific XML vocabulary widely used for technical documentation. There are widely-available tools to convert documents between DocBook and a wide variety of other formats, including ODF (q.v.) and into PDF.

DocBook is a fairly large vocabulary, and uses the “element pool” concept described by Eve Maler and Jeanne El Andaloussi in *Developing SGML DTDs* to make authoring easier: a core set of element names are available pretty much everywhere.

Sometimes people criticize DocBook because the most common conversion to HTML does not produce elegant pages, but that is not a feature of DocBook itself, and when the conversion goes well, people do not realize the original format was DocBook.

10. Customizing DocBook: Mallard

DocBook from the beginning provided for extensions, using an SGML and XML DTD syntax to allow people to add their own elements and modules. But sometimes you want to *reduce* rather than *extend*.

A difficulty for authors using DocBook can be that at any given point in a document there can be a bewilderingly large number of possible elements that can be inserted. People working with DocBook all day quickly learn them, but occasional users can find this to be a barrier. Mallard is a subset of DocBook that was devised for writing online help the GNOME project, and is an example of a customization. Mallard also has its own Markdown variant, *Ducktype*. Mallard also adds semi-automatic linking between topics, so that some processing is required before the files are usable by DocBook tools, but that is not an issue in the context of online help for the GNOME desktop.

Note: Although DocBook has its own extension mechanisms, Mallard ended up as a standalone system, and today uses elements from HTML such as *p* and *em* rather than *para* and *emphasis* of DocBook. Like many project-specific extensions, it evolved.

Starting off with an industry-standard vocabulary and customizing it can considerably reduce the work needed to develop a format, but people within the project have to make a conscious decision about the value of having their documents conform to the original vocabulary.

11. The Text Encoding Initiative (TEI)

The Text Encoding Initiative is both an organization and an eponymous vocabulary and associated methodology for transcribing texts for the purpose of analysis, study, and the production of fac simile editions. It is widely used in the Digital Humanities.

The TEI vocabulary was originally intended *always* to be customized to particular academic projects. However, a core subset, TEI Lite, became popular. The TEI Consortium provides an extension mechanism, One Document Does It

All (ODD), which simplifies making a customization of TEI, and in particular one that includes and combines specific "modules" such as one for drama or one for transcribing natural-language dictionaries.

The primary context for using TEI is academic text processing and study of texts. An advantage of using it in that context is the high level of expertise and availability of assistance from other academics, as well as tools

12. Open Document Format (ODF), OOXML

Whereas PDF files are compressed binary extensions of the PostScript language which, when executed, produce page images, ODF and OOXML are XML representations, again binary and compressed but extractable as text. However, rather than specifying the position of items on the page, ODF and OOXML provide an XML-based representation of editable documents.

Word processors generally are strong in their tools for commenting on documents and performing collaborative reviews. The model is that a reviewer sends an annotated copy of a document to the author, who then in turn reviews the comments and suggested changes; this model fits well with many social contexts of writing and working with documents.

The ODF and OOXML formats were designed to serialize OpenOffice and Microsoft Word™ documents (respectively) to XML, and support document revision. Unlike PDF, ODF and OOXML documents are intended to be editable by the recipient, and text reflows.

Unfortunately, the OOXML specification, despite being very large, is also incomplete. In addition, the XML is written in the word-processing implementation domain, not in the user's problem domain. The XML is complex and relatively difficult to work with, although freely available libraries and XSLT transformations exist to work with them. Word processor formats tend not to nest structures very much: for example, list items tend to be considered as automatically-numbered paragraphs and not to be nested inside a containing list element. Points at which there had one been style changes or selection boundaries may also be retained in the markup, further complicating processing.

Word processing files are not generally ideal for archiving or interchange except in specific contexts such as the need for review or specific print-based workflows. Generated HTML may also have accessibility problems, and unless users are systematic with the use of named styles there tends to be only weak semantic labeling. Word processors in many cases reduce the apparent cost of writing, because they make it seem easy. Unfortunately in a wider context this can merely result in increasing costs and complexity elsewhere in workflow processes when finer-grained control over markup and document features may be needed. In this regard, word processor files can be similar to HTML documents. In all cases final archived documents need to be saved in format that is independent of any particular version of any software.

13. Raster Images

Although raster (bitmap) image formats such as TIFF, PNG, GIF ,BMP, JPEG and so forth can be suitable for non-textual content, text inside such image formats is really just a picture of text: it cannot be searched or copied and pasted, and text readers are not able to speak it out loud.

Note that JPEG images in particular use a *lossy* compression, which introduces "artifacts" that can be visible and can hinder optical character recognition or other reuse of the files.

Document page-images are most often used in situations where digital versions of original documents are unavailable and paper copies have been scanned. For archival use it's important to use an image format that is open and standard and that has as few interoperability problems as possible. PNG is better than TIFF in this regard, since TIFF files vary between platforms, but both are better than Windows BMP or PhotoShop PSD (proprietary) or JPEG (lossy).

14. Conclusion

The purpose of this paper is to discuss some formats, both XML and non-XML, with a perspective of the contexts in which each format is appropriate. There is no single "best" format, and no single way to represent information digitally. Rather, the context in which information will be used determines the perspective from which a choice should be made.

Bibliography

[DB5SPEC] Norman Walsh: The DocBook Schema. Working Draft 5.0a1, 29 June 2005, OASIS. http://www.docbook.org/specs/wd-docbook-docbook-5.0a1.html

[SGMLDTD] Eve Maler, Jeanne El Andaloussi: Developing SGML DTDs. 15 December 1995,Prentice Hall PTR

Validating `selector`

Syd Bauman, Northeastern University Digital Scholarship Group

Abstract

I needed a way to validate that the value of `tei:rendition/@selector` was a proper CSS3 *selector*. So I wrote a regular expression to do so. An 18,385-character-long regular expression. And it seems to work.

1. Introduction

Starting with P3 in 1994 (i.e., over two years before CSS1 was released), the Text Encoding Initiative *Guidelines for Text Encoding and Interchange* supported a mechanism to indicate a default rendition, a way of saying "all `emph` elements were in bold italics in the original." The method used to indicate with which element type a particular default rendition was associated was to give the element type as the value of the `gi` attribute of the `tagUsage` element. The value of this attribute could be validated in broad strokes by giving it a datatype of teidata.xmlName (which boils down to xsd:NCName). Furthermore, it could be checked to be of an element type that occurs in the document using some simple Schematron:

```
<sch:let name="instanceTypes"
         value="distinct-values( //tei:TEI/tei:text//tei:*/local-name() )"/>

<sch:pattern>
  <sch:rule context="tei:tagUsage[@gi]">
    <sch:assert test="@gi = $instanceTypes">
      @gi should contain the name of an element that is within the
      &lt;text> of the document.
    </sch:assert>
  </sch:rule>
</sch:pattern>
```

Starting in 2015-10 with [P5] [2.9.0], TEI introduced a new method for the same purpose (and then phased out the original method). In this new method, rather than simply giving the element type to which a default rendition applied, a user specifies to which elements a default rendition applies using the CSS selection mechanism. This allows far greater flexibility and precision in expressing to which instance elements a default rendition applies, at very little to no cost in processing when using CSS to directly render TEI. For example, it is quite common in early modern printed books to have the signatures [https://en.wikipedia.org/wiki/Signature_mark] centered on the bottom of certain pages, and the catchwords [https://en.wikipedia.org/wiki/Catchword] on the bottom right of each page. Each of these phenomena is encoded in TEI using the `fw` [https://tei-c.org/Vault/P5/3.5.0/doc/tei-p5-doc/en/html/ref-fw.html] element, but with different values of its `type` attribute. Thus it would not be surprising to find the declarations in Example 1 [188] in a `teiHeader`.

Example 1. Sample **`rendition`**s

```
<rendition selector="fw[type='sig']">text-align: center;</rendition>
<rendition selector="fw[type='catch']">text-align: right;</rendition>
```

However, there is a significant cost to this improvement in the system with respect to our ability to validate.[1] The TEI only defines `selector` as teidata.text (which boils down to the RELAX NG string datatype).

This struck me as insufficient, and when I found a simple syntactic error in a `selector` in one of our textbase files, I decided to try to improve on the situation. The TEI does not say from which version of CSS the selector syntax should be taken, so I chose level 3 ([sel3][2]).[3] The only formal constraint system available in the TEI schema language[4] above and beyond enumerated lists of values and XSD datatypes is the W3C regular expression language. Thus I set about writing a regular expression to validate CSS3 selectors.

2. Against all odds

According to several sites and Stack Overflow answers,[5] the CSS3 language is not regular, and *cannot* be parsed with a regular expression. So how was I able to do this? I think there are three contributing factors.

[1]It is also costly to use this system when one wishes to convey the indicated renditions when converting the TEI to some other markup language, e.g. XHTML or ePUB. That is a topic for another paper, though.

[2]Since I did this work, a newer version, [sel3N], has been released. The new version seems at first blush to be substantially the same as the one I was using; section 4 *Selector syntax* is word for word identical.

[3] I was at the time blissfully ignorant of [sel4], which is still in Working Draft.

[4]Pure ODD. See various discussions including a brief overview [https://wiki.tei-c.org/index.php?title=ODD#.22Pure_ODD.22], the TEI introduction [http://www.tei-c.org/Support/Learn/odds.xml], an *Extreme Markup Languages* paper on ODD [http://conferences.idealliance.org/extreme/html/2004/Burnard01/EML2004Burnard01.html], a *Journal of the TEI* paper on Pure ODD [http://jtei.revues.org/842], the specification [http://www.tei-c.org/release/doc/tei-p5-doc/en/html/TD.html], or the description of ODD processing [http://www.tei-c.org/release/doc/tei-p5-doc/en/html/USE.html#IM].

[5]For example, https://stackoverflow.com/a/12575871/9741160, https://stackoverflow.com/a/12126644/9741160, and of course section 10.1 of [CSS3].

subset of strings. First, I was not dealing with the entire CSS3 language, but with only a distinct subset, selectors. That said, section 10.1 of [CSS3] asserts that even the selector syntax alone is an LL grammar, which implies it is not regular.

subset of task. I did not need to actually process the selector — I did not need to determine to which instance elements in the document the specified default rendition should apply — I merely needed to know whether a CSS3 processor *could* process the selector. Thus I had no need to parse selectors into their component segments, but rather had only the somewhat simpler task of ascertaining if a given string is a proper CSS3 selector or not.

subset of knowledge. Perhaps most importantly, I did not know it was impossible until after I'd done it. If I had read that CSS3 could not be matched by a regular expression before I had tried this, I may not have been smart enough to think "well, I only need the selector bit; and I don't need to parse it, only to validate, so maybe a regular expression will do." Quite likely I just would not have tried.

> Some of the world's greatest feats were accomplished by people not smart enough to know they were impossible.
>
> —Doug Larson

3. Writing the Regexp

CSS3 selectors are complex. For example, `blockquote > div p`[6], `div.stub *:not(:lang(fr))`[7], `*|*[a|foo~="bar"], *|*[|class~="bar"]`[8], and `stub ~ [|attribute^=start]:not([|attribute~=mid])[|attribute*=dle][|attribute$=end] ~ t`[9] are all valid CSS3 selectors. And while these are probably somewhat complicated for real-life applications, they are simple compared to what a CSS3 selector *could* be.

So how does one write a regular expression for something this complex? The answer, of course, is rather than trying to write the regular expression directly, you write a program to generate the regular expression. I have used this approach in the past, finding that it is generally not too difficult to manually convert a small EBNF grammar or other set of formal rules into a small program to generate a corresponding regular expression.[10] Typically each non-terminal becomes a variable, defined in terms of constants (for the terminals) and the variables that have been defined so far (for the non-terminals).

As a trivial example, Example 2 [189] is a small Perl program that generates a POSIX extended regular expression that matches an integer, as defined by the EBNF provided in the Wikipedia page on EBNF [https://en.wikipedia.org/wiki/Ebnf].[11]

Example 2. A Perl program that generates a regular expression

```
#!/usr/bin/env perl
#
# Copyleft 2019 Syd Bauman and Northeastern University Digital
# Scholarship Group.
#
# No parameters; reads no input. Writes out a regular expression
# that matches an integer, where integer is defined by the EBNF
# in https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form:
# | digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
# | digit                = "0" | digit excluding zero ;
```

[6] Nondeterministic matching of descendant and child combinators [https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/xml/full/flat/css3-modsel-86.xml]
[7] NEGATED :lang() pseudo-class [https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/xml/full/flat/css3-modsel-67.xml]
[8] Attribute space-separated value selector with declared namespace [https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/xml/full/flat/css3-modsel-99.xml]
[9] Dynamic handling of attribute selectors [https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/xml/full/flat/css3-modsel-d3.xml]
[10] Of course, since an EBNF grammar can represent any context-free language (Chomsky Type 2), there are some EBNFs that cannot be represented by a regular language (Chomsky Type 3), although some regular expression languages (e.g., PCRE) have extensions that allow them to represent any context-free grammar.
[11] Readers who are well versed in PCRE will know that the EBNF can be represented directly in the regular expression, e.g.:

```
(?(DEFINE)
        (?<digit_sans_zero> (1|2|3|4|5|6|7|8|9)            )
        (?<digit>           (0|(?&digit_sans_zero))        )
        (?<natural_number>  (?&digit_sans_zero)(?&digit)* )
        (?<integer>         (0|(-?(?&natural_number)))     )
        )^(?&integer)$
```

While this is impressive, and very useful in its own right, it is not helpful to me here as I am interested in generating a W3C regular expression, not in using PCRE.

```
# | natural number = digit excluding zero, { digit } ;
# | integer = "0" | [ "-" ], natural number ;
# The resulting regexp is intended to be a POSIX ERE, but would
# also work as a PCRE or a W3C regular expression, and probably
# lots of others. (But not a POSIX BRE or an Emacs LISP regexp.)

$digit_sans_zero = "(1|2|3|4|5|6|7|8|9)"; # could be just "[1-9]" :-)
$digit = "(0|$digit_sans_zero)";
$natural_number = "($digit_sans_zero($digit)*)";
$integer = "(0|(-?$natural_number))";

print STDOUT "$integer\n";
exit 0;
```

While I am sure there has been much written on this general approach,[12] I was not looking for general-purpose (regular) grammar to regular expression conversion, I was just looking to convert a particular grammar to a regular expression.

3.1. Generating the generator

Thus I fell back on old habits, and began writing what I thought would be a short routine to write a moderately long regular expression. Because I had used Perl for this in the past, and because Perl is interpreted (and thus an easy language with which to perform rapid cycles of tweak-and-test), I wrote this program in Perl. This, I now believe, was a mistake.

The immediate output of the program was to be a regular expression — that is, a string — which I imagined I would generate once (after building and debugging the generation routine) and copy into an appropriate schema. Thus a string manipulation language like Perl seemed appropriate. However, in my zeal I forgot a universal truth about writing programs, even simple ones: they need to be *tested* and *debugged*, *repeatedly* and *thoroughly*. In this case each round of testing required that the string be copied from standard output into a schema against which some test data could be validated. (Remember that I could not use Perl to directly test the regular expression against test data, because I was not generating a Perl-flavored regular expression, but rather an W3C-flavored regular expression.) Thus in order to save time, it made sense for the program to either insert the regular expression into the test schema for me, or to write a complete test schema (that includes the regular expression) anew each time it was run. While the former technique is perhaps more desirable from a point of view of separation of concerns, the latter is much easier to write and is preferable insofar as it keeps all the concerns (as it were) in one file.

My preferred schema languages are RELAX NG and ISO Schematron,[13] either of which can be used to test the value of `tei:rendition/@selector` against a W3C-flavored regular expression. Thus I soon modified the generation program so that instead of writing just a regular expression to standard output, it wrote a small, but complete RELAX NG schema or a small, but complete, XSLT program, either of which was designed to test only the value of `selector` against the (current version of) the generated regular expression.

The reason for generating XSLT output instead of ISO Schematron output was purely pragmatic. The Schematron processor I use works by converting the Schematron to an XSLT intermediate (using XSLT), and then transforming the test document using the intermediate XSLT. By writing XSLT directly from the generation program, I could save a conversion step during each test and still use the same engine to execute the regular expression.

Details about the design of the output RELAX NG schema and XSLT program follow. But their mere existence explains why my use of Perl was a mistake. Both of the desired output formats were XML, and for me XSLT is the best language to use for generating XML as output.[14] (Even those who do not think of XSLT as the *best* language for writing XML will admit that it is far better than Perl.)

3.2. Case

If there is a method of asking a RELAX NG validator to use a regular expression case insensitively, I do not know it. Thus the regular expression is written case sensitively. E.g., the sub-pattern `[A-Za-z]` occurs frequently where `[A-Z]` would be acceptable if the pattern could be applied after case folding.

[12] A quick web search demonstrates that discussions of this topic come in all flavors, from stackoverflow posts to class slides, to full academic papers. See, e.g. [FA2RE], [TPoRE], [REBNF] [NFA2RE], [DFA2RE], and [REGGE].

[13] That is, my preferred schema languages other than TEI PureODD.

[14] I believe it is very advantageous to use a language, like XSLT, that outputs a *tree* in serial format as XML — rather than as a sequence of characters, some of which are pointy brackets — and thus *cannot* make most well-formedness errors. Without such a language, simple well-formedness errors creep in constantly. Even in the 297 XML files that make up the XML version of the W3C CSS3 Selectors Test Suite Index [https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/], which look to me like they are generated by a program, I found four files with one well-formedness error each. (“`<br>`” without an end-tag in all four cases).

It is slightly advantageous to be absolutely explicit about which characters are allowed, so in one sense this verbosity is an advantage. On the other hand, there are two significant disadvantages:

1. It adds verbosity. The generated regular expression is more than 1000 characters longer than it would be if case insensitivity could be assumed.
2. It means that the generated regular expression is in some cases technically incorrect. For example, [CSS3] defines a pseudo-class `:link`. It never mentions a pseudo-class `:LINK`, and I have never seen it used in uppercase in the real world or in a test suite. However, section 3 says quite clearly "All Selectors syntax is case-insensitive within the ASCII range (i.e. [a-z] and [A-Z] are equivalent)". Thus either matching should take place case insensitively, or wherever the regular expression says `link` it should really say `[Ll][Ii][Nn][Kk]`.

3.3. Language

The first and only parameter accepted by the CSS3 `:lang()` pseudo-class *does not* need to be a valid language tag per [BCP47]; it only needs to be a valid CSS 2.1 identifier. However, because BCP 47 is the system used by TEI to indicate language (on `xml:lang`), use of other language identifiers in this context does not make sense. Thus the generated regular expression requires a BCP 47 language tag as the parameter to `:lang()`.

Rather than re-invent this particular wheel, I guessed that others had already written a regular expression that would match a BCP 47 tag. Indeed I found more than one readily available on the web. The program currently uses a regular expression adapted from one made publicly available by its author, Seb Insua, a consultant software engineer based in London.

3.4. What the generator generated

The Perl generation program is called, somewhat unimaginatively, CSS3_selector_regex_generator.perl. As mentioned above, while its primary output is conceptually a long regular expression, in practice the primary output is that regular expression in the context of either a RELAX NG schema (XML syntax) or an XSLT program designed just to test the regular expression. In both cases, the schema could have been designed to test only that which was the target of this entire endeavor: the `tei:rendition/@selector` attribute. But I chose instead to have the output schemas test *any* `selector` attribute, for reasons explained in Section 3.5 [192], below.

3.4.1. RELAX NG schema

The output RELAX NG schema is designed to constrain only the value of `selector`, nothing else. Thus it allows any outermost ("root") element from any namespace (including the null namespace), which is allowed to have any attributes from any namespace (including the null namespace) and any children including text intermingled with any number of any element from any namespace (including the null namespace); each child element in turn is also allowed to have any attributes from any namespace (including the null namespace) and any children including text intermingled with any number of any element from any namespace (including the null namespace), *except* that any `selector` attribute on any element must match the generated regular expression. Thus the only error messages generated by validation against the schema assert that the value of a `selector` attribute is not a CSS3 selector.[15]

The main declaration of the RELAX NG schema is shown in Figure 1 [191]; the placeholder *`GENERATED_REGEXP_HERE`* indicates where in the schema the generated regular expression is placed. The schema is shown in the compact syntax although it is generated in the XML syntax.

Figure 1. RELAX NG schema constraint

```
start = ANY
ANY =
  element * {
    attribute * - selector { text }*,
    attribute selector {
      xsd:string {
        pattern = "GENERATED_REGEXP_HERE"
      }
    }?,
    (text | ANY)*
  }
```

[15] Or, of course, messages indicating that the input document is not well-formed XML, but for our purposes here those are messages from a pre-validation XML parser, not from the validator itself.

3.4.2. XSLT “schema”

The generated XSLT stylesheet used as a schema is deliberately more verbose. It generates a single line of text output for each occurrence of `selector` in the input which indicates whether or not the value of said `selector` matches the generated regular expression or not. Figure 2 [192] shows a pared-down version of the generated XSLT stylesheet. Again, *`GENERATED_REGEXP_HERE`* indicates where in the output the generation routine places the regular expression.

Figure 2. XSLT “schema”, validation

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0">

  <xsl:variable name="selector_regex">
    <xsl:text>GENERATED_REGEXP_HERE</xsl:text>
  </xsl:variable>
  <xsl:variable
    name="anchored_selector_regex"
    select="'^'||$selector_regex"/>

  <xsl:output method="text"/>

  <xsl:template match="/">
    <xsl:text>&#x0A;</xsl:text>
    <xsl:apply-templates select="//*[@selector]"/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:value-of select="'selector “'
                          ||@selector
                          ||'” is&#x09;&#x09;'"/>
    <xsl:if
      test="not( matches( @selector, $anchored_selector_regex ) )">NOT </xsl:if>
    <xsl:value-of select="'valid.&#x0A;'"/>
  </xsl:template>

</xsl:stylesheet>
```

3.5. Rapid cyclic debugging

With the generation of a complete RELAX NG schema or XSLT program instead of just a string, I could quickly and easily test that the regular expression generated was at least a valid regular expression just by validating an XML document (any XML document) against the generated RELAX NG schema or transforming an XML document (any XML document) with the generated XSLT stylesheet. In either case, the processor (for me that generally means jing or Saxon) will generate an error message if the string being used as a regular expression is not in fact parsable as a W3C regular expression. For example, for the string “This is)bad(” jing will generate an invalid parameter: invalid regular expression: character is not allowed in this context: This is >>>>)bad(message, and Saxon a Syntax error at char 8 in regular expression: Unmatched close paren message.

Testing that a valid regular expression works as desired is the next step. In a practical sense, this is quite easy: just take an XML file that has a test selector on a `tei:rendition/@selector`, and validate it against the generated RELAX NG schema or transform it with the generated XSLT stylesheet. But, in a logical sense, this is quite difficult: what selectors get tested? What strings that are not selectors get tested?

3.5.1. Practical

As mentioned above, rapid cyclic testing is not particularly difficult: just take an XML file that has a test selector on a `tei:rendition/@selector`, and validate it against the generated RELAX NG schema or transform it with the generated XSLT stylesheet. But I am personally fond of creating self-testing systems, in part because it takes literate programming to a next step, keeping the test cases in with the original documentation and program, thus keeping all the concerns (as it were) in one file; and in part just because it’s cool.

Observing that

- RELAX NG permits elements from namespaces other than the RELAX NG namespace to occur in a grammar (even as a direct child of `rng:grammar`), and
- XSLT permits elements from namespaces other than the XSLT namespace to occur in a stylesheet (even as a direct child of `xsl:stylesheet` or `xsl:transform`),

I could insert the desired tests directly into the `CSS3_selector_regex_generator.perl` program such that they would be inserted into the output RELAX NG schema or XSLT program as the value of a `tei:rendition/@selector`. The regular expression could then be tested against the test CSS selectors by validating the output RELAX NG grammar *against itself* (that is, use the generated RELAX NG schema as both the grammar and the document instance), or by transforming the XSLT stylesheet *with itself* (that is, use the generated XSLT program as both the stylesheet and the input document).

3.5.2. Logical

This self-testing system worked quite well while I was in initial development of the regular expression and various sub-portions thereof, but pretty quickly (well, not that quickly — it took longer than I care to admit to move from initial development to refinement) it became necessary to test the regular expression against an array of possible selectors. Luckily

- lots of CSS files, using a variety of selectors of varying complexity, are readily available both at my own project and on the web, and
- there are several CSS test suites, including one from the W3C, available on the web.

So, in addition to the few dozen scenarios I had dreamt up, I incorporated thousands of selectors from real CSS files and test suites. Since my tests would (deliberately) examine any `selector` attribute, not just one on a `tei:rendition` element, I was even able to make test cases consistent (each test on an `selector` of a `rendition`) and simultaneously retain the provenance of each test by using an appropriate namespace for the `rendition`.

Example 3 [193] is an extract of the generated XSLT showing a few of the namespaces used, and three test `selector`s for each of those namespaces. The actual generated XSLT includes the validation portions discussed above, and tens to hundreds of test `selector`s for each of over a half dozen namespaces.

Example 3. XSLT “schema”, debugging

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:sb="http://bauman.zapto.org/ns-for-testing-CSS"
  xmlns:wpt="https://github.com/web-platform-tests/wpt"
  xmlns:w3c="https://www.w3.org/Style/CSS/Test/CSS3/Selectors/current/"
  xmlns:wo="http://wwo.wwp-test.northeastern.edu/WWO/css/wwo/wwo.css"
  xmlns:pt="https://github.com/benfrain/css-performance-tests"
  version="3.0">

  <!-- This pgm written 2019-06-02T09:57:25 by
       ./CSS3_selector_regex_generator.perl -->

  <!-- ========= debugging ========= -->
  <!--
  legend:
    pt = performance test suite
    wpt = W3C web platform tests for CSS
    w3c = W3C test suite, last retrieved 2019-06-01
    wo = WWO CSS stylesheet, i.e. for Women Writers Online textbase viewing site
    sb = dreamt up by yours truly
  -->
  <!--
  Note: the wpt and w3c sets are very very similar, but not quite
    identical; it is not clear to me there is any real advantage in
    running both, but I am interested in having a lot of test cases
    too see how fast this is, too. Thus both are included.
    -->
```

```
  <!-- ********* -->
  <wpt:rendition selector="li,p "/>
  <wpt:rendition selector="p "/>
  <wpt:rendition selector="p[title$='bar'] "/>
  <!-- ********* -->
  <w3c:rendition selector='   stub ~ [|attribute^=start]:not([|attribute~=mid])
                         [|attribute*=dle][|attribute$=end] ~ t '/>
  <w3c:rendition selector='   #two:first-child '/>
  <w3c:rendition selector='   #three:last-child '/>
  <!-- ********* -->
  <pt:rendition selector='[data-select]'/>
  <pt:rendition selector='a[data-select]'/>
  <pt:rendition selector='[data-select="link"]'/>
  <!-- ********* -->
  <sb:rendition selector=":not(:lang(en))"/>
  <sb:rendition selector=":not( :lang(   en-GB ))"/>
  <sb:rendition selector="    :lang(en-GB-x-HPf)"/>
  <!-- ********* -->
  <wo:rendition selector="#popup > div.note.content .bibl-sref "/>
  <wo:rendition selector='#popup > div.note.content .bibl-sref span[class~="moo"],
#popup > div.note.content .bibl-sref-parenless span[class~="moo"] '/>
  <wo:rendition selector=""/>
  <!-- ========= end debugging ========= -->

</xsl:stylesheet>
```

4. Resources

4.1. Time

I was quite worried that this regular expression would take a long time, perhaps even forever, to run. I was pleasantly surprised to find it could be quite speedy. In a test run using the RELAX NG grammar, jing tested over 5900 `selector` attributes in under ½ second; well under 0.1 ms each. The typical TEI file will only have a half dozen.

XSLT was impressive, but in the other direction. In one typical test of transforming the generated XSLT with itself, with only 40 `selector` attributes in it, saxon9he took almost 03:49, or over 175 ms each.

At the Women Writers Project we currently have 1,685 `selector` attributes in 449 files, with a range of 0 to 13 `selector`s per file. Our encoders have been using a schema which incorporates this regular expression for the past 9 months, and no one has complained about speed. Note that our encoders use oXygen with Enable automatic validation on and set to a delay of 1 s.

4.2. Memory

Whether using jing to validate the RELAX NG schema against itself, or Saxon to transform the XSLT stylesheet with itself, the java virtual machine required more RAM than the defaults set on my machine. At 4 MiB of stack space (`-Xss4m`) jing ran out; at 8 MiB it did not. Saxon did not need extra stack space, but craved an extraordinary quantity of heap space. Astoundingly (at least to me), at 5 GiB of heap space (`-Xmx5g`) it bombed; with 6 GiB it ran.

4.3. Storage

The regular expression itself is very large in terms of a string (18,385 characters as of this writing). Thus it is quite cumbersome and somewhat annoying to manage in almost any file, be it schema, XSLT, or the list of error messages that helpfully say must match ….

However, in terms of actual storage space, the `CSS3_selector_regex_generator.perl` program, its RELAX NG output, and its XSLT output all together take up less than 1.7 MiB on my filesystem. That is, they could *almost* fit on a 1980s 3½ inch floppy disk.

5. Future work

- I plan to convert the generation program to XSLT in August and early September of 2019.

- Right now testing is not performed rigorously enough — while the regular expression is tested against thousands of selectors including the entire W3C test suite, the results are simply checked by human eye. This should be automated. My current thought is to add a `selector_is_valid` attribute whose value is one of `true`, `false`, or (before being verified, or perhaps in some bizarre cases) `unknown`.[16]
- I plan to present the regular expression itself and the XSLT generation program for it at the TEI Members' Meeting and Conference in Graz, 2019-09, with the aim of convincing the TEI Consortium to use the regular expression in the TEI schema.

6. Availability

The current Perl program is available under the GPL in the WWP public code share repository [https://github.com/NEU-DSG/wwp-public-code-share/]. I expect (or at least hope) to replace it with an XSLT version in August or September of 2019.

Bibliography

[P5] The TEI Consortium, TEI P5: Guidelines for Electronic Text Encoding and Interchange. https://tei-c.org/guidelines/p5/

[2.9.0] The TEI Consortium, TEI P5: Guidelines for Electronic Text Encoding and Interchange, Release 2.9.0 [https://tei-c.org/Vault/P5/current/doc/tei-p5-doc/readme-2.9.1.html] .

[FA2RE] Alexander Meduna, Lukáš Vrábel, and Petr Zemek Converting Finite Automata to Regular Expressions http://www.fit.vutbr.cz/~izemek/grants.php.cs?file=%2Fproj%2F589%2FPresentations%2FPB05-Converting-FAs-To-REs.pdf&id=589

[TPoRE] Nikita Popov The true power of regular expressions https://nikic.github.io/2012/06/15/The-true-power-of-regular-expressions.html

[REBNF] Michael Wollowski Regular ExpressionsBackus-Naur Form (BNF) https://www.rose-hulman.edu/class/se/csse404/schedule/day2/02-REBNF.pdf

[NFA2RE] Convert NFA to regular expression https://girdhargopalbansal.blogspot.com/2013/06/convert-nfa-to-regular-expression.html

[DFA2RE] DFA to Regular Expression | Examples Akshay Singhal https://www.gatevidyalay.com/dfa-to-regular-expression-examples-automata/

[REGGE] Regular expression generation through grammatical evolution Ahmet Cetinkaya https://dl.acm.org/citation.cfm?id=1274089

[CSS3] Selectors Level 3 W3C Recommendation 29 September 2011 W3C http://www.w3.org/TR/2011/REC-css3-selectors-20110929/

[CSS3new] Selectors Level 3 W3C Recommendation 06 November 2018 W3C https://www.w3.org/TR/2018/REC-selectors-3-20181106/

[sel4] Selectors Level 4 W3C Working Draft, 21 November 2018 W3C https://www.w3.org/TR/2018/WD-selectors-4-20181121/

[BCP47] Tags for Identifying Languages IETF https://tools.ietf.org/html/bcp47

[16] That is, a teidata.xTruthValue.

XSpec in the Cloud with Diamonds

Sandro Cirulli, XSpec

Abstract

Running XSpec tests in a development team is usually performed via a CI server/service. However, this comes with limitations related to the use of private repositories and to the cost and burden of administering CI servers.

This paper describes an alternative approach for running XSpec tests from private repositories using a serverless architecture built on AWS Lambda. It describes the technical configuration and discusses the benefits, cost optimization, and constraints of a serverless architecture for running XSpec tests.

1. Introduction

XSpec is a unit test and behaviour-driven development framework for XSLT, XQuery, and Schematron [xspec] [203]. XSpec test suites are generally executed:

- On a local machine using the shell or batch scripts, the oXygen XML editor, etc.
- On a Continuous Integration (CI) server using software tools like Jenkins or online CI services like Travis, CircleCI, AppVeyor, etc.

While running tests locally is typically performed by individual developers during their development process, running tests on a CI server is usually employed within a team to integrate code changes under version control.

CI servers like Jenkins offer fine grained control on how the test suite is executed and support both private and public version controlled repositories. However, this configuration requires a server to run the CI software and a system administrator to maintain both the server and the software.

Conversely, online CI services like Travis run entirely on the cloud and do not require system administration or server maintenance. On the other hand, these CI services are free to use only for open source projects in public repositories and charge fees for private repositories and improved capabilities.

The aim of this paper is to show an alternative approach using a serverless architecture built on AWS Lambda. This allows to run XSpec tests from private repositories while keeping costs low and avoiding server and software maintenance.

2. AWS Lambda and Serverless Architecture

Lambda is a compute service provided by Amazon Web Services (AWS) allowing to run code without provisioning or managing servers [lambda] [203]. AWS Lambda claims to scale applications and workloads automatically from few requests per day to thousands per second and charges only for the compute time used.

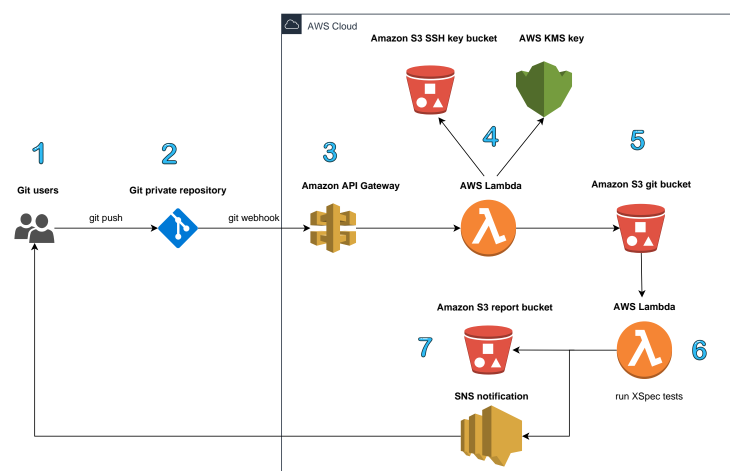
AWS Lambda can be used in conjunction with other services to build a serverless architecture. This is a cloud architecture typically running a function inside a stateless computing environment triggered by an event [fowler] [202]. Serverless architectures enable to run applications and workloads without managing server infrastructure and with reduced operational costs and flexible scaling [cncf] [202].

In the next section I am going to explain how to build a serverless architecture for running XSpec test suites triggered by an event such as pushing new code to a version control system.

3. Technical Configuration

Figure 1 [198] illustrates the workflow for running XSpec tests in a serverless architecture.

Figure 1. Workflow for running XSpec tests in a serverless architecture



The workflow comprises the following steps:

1. A developer pushes a code change into a private git repository.
2. A git webhook sends a payload to an Amazon API gateway.
3. The API gateway endpoint accepts the webhook request from git and triggers a Lambda function.
4. The Lambda function connects over SSH to the git service. SSH private keys are stored securely using Amazon S3 and AWS KMS.
5. The zipped content of the git repository is stored in S3.
6. The S3 storage event triggers a lambda function that unzips the content of the git repository and executes XSpec tests.
7. The report of the XSpec tests is stored in S3. A notification is sent via email through SNS to the development team.

For the readers unfamiliar with AWS, I shortly describe the purpose of each Amazon service mentioned in the diagram and in the next sections:

- Amazon API Gateway: a managed service allowing to create API endpoints [api_gateway] [202].
- Amazon S3: an object storage service (S3 stands for Simple Storage Service) [s3] [203].
- Amazon KMS: a managed service for creating, storing and accessing encryption keys (KMS stands for Key Management Service) [kms] [203].
- AWS Lambda: a service for building serverless applications and running code without managing servers [lambda] [203].
- Amazon SNS: a messaging service for sending notifications such as emails, sms, etc. to subscribing clients (SNS stands for Simple Notification Service) [sns] [203].
- AWS CloudFormation: a service for describing and provisioning infrastructure resources in AWS [cloudformation] [202].
- AWS IAM: a service for controlling access to AWS resources (IAM stands for Identity and Access Management) [iam] [203].
- Amazon CloudWatch: a service for monitoring AWS resources [cloudwatch] [202].
- AWS Billing and Cost Management: a service for paying, monitoring, and budgeting costs in AWS [billing] [202].

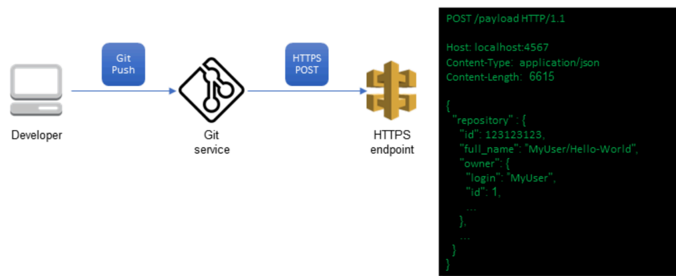
In the next sections I am going to describe how to set up and configure the serveless architecture. First, I am going to link the private git repository with S3 in order to store changes whenever a user pushes new commits to git (steps 1 to 5). Then I'm goint to configure the lambda function to run XSpec tests and send notifications to the user (steps 6 and 7). Whenever possible, I will refer to code in my GitHub account so that readers wishing to replicate this can get hold of code examples and templates.

3.1. Linking Git to S3

The easiest way to link git repositories to S3 and build the first part of the infrastructure in the diagram is to use the Quick Start deployment guide provided by AWS [git2s3] [203]. This contains a quick start button allowing to start all the necessary AWS resources with a single click using an AWS CloudFormation template. The process is thoroughly documented in the guide which also describes several options for customizing the configuration.

In my GitHub repository [markupuk2019] [203] I wrote additional documentation for my configuration. In particular, I set up a GitHub private repository and used a GitHub webhook to send a payload to the AWS API Gateway whenever a new commit is pushed to the GitHub repository (this option is referred to as git pull endpoint in the Quick Start guide). Figure 2 [200], taken from the official Quick Start documentation, illustrates the webhook workflow.

Figure 2. Workflow for triggering the webhook



This configuration covers points 1 to 5 of the diagram. At the end of the process the GitHub private repository is replicated in a zip file within S3.

3.2. Lambda Configuration

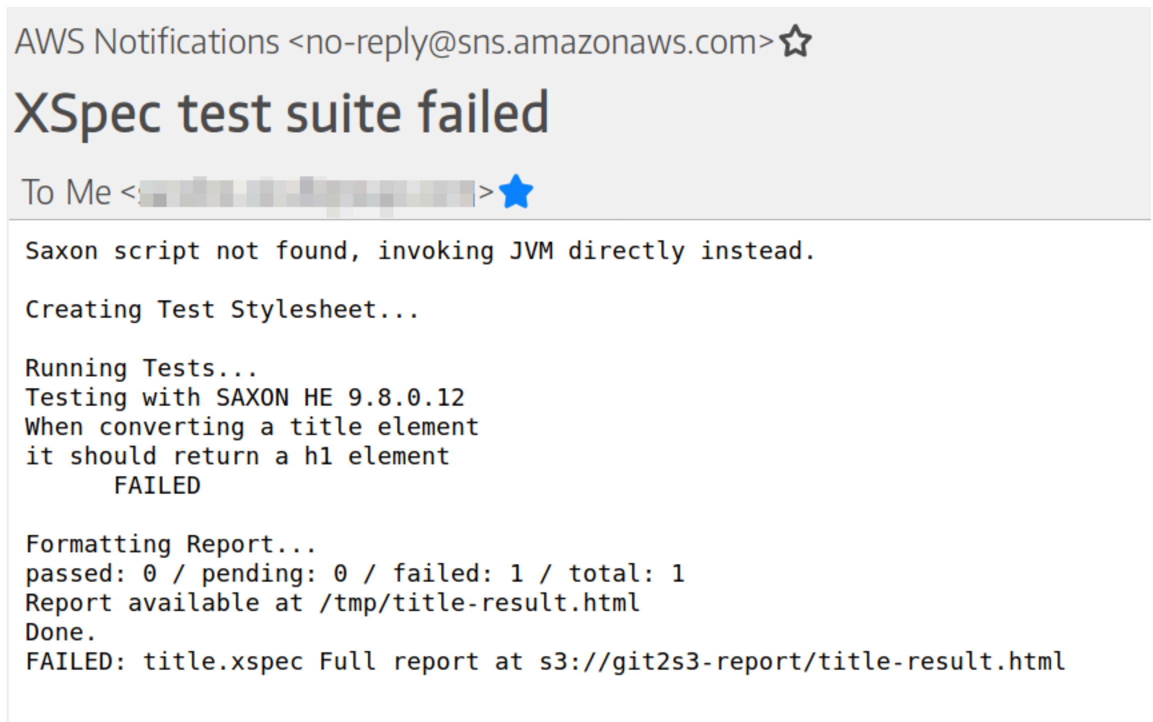
This configuration covers point 6 and 7 of the diagram. In particular, it describes the configuration for setting up the lambda function running XSpec tests. This entails:

1. Retrieving the GitHub code stored in a zip file in S3.
2. Unzipping the file and storing it in a local directory accessible by the lambda function.
3. Setting up environment variables required by XSpec (i.e. paths to Saxon and HTML report).
4. Running the XSpec test suite for all the XSpec tests stored in a given directory.
5. Storing the HTML report in S3.
6. Notifying the developer in case a test failed and providing a link for accessing the HTML report in S3.

A step-by-step guide to replicate this configuration and the code for the lambda function is available in my GitHub account [markupuk2019] [203]. It is worth highlighting the following points:

- Custom Runtime: AWS Lambda natively supports lambda functions written in Java, Go, PowerShell, Node.js, C#, Python, and Ruby. AWS Lambda also provides a Runtime API allowing to implement the lambda function in any programming language [custom_runtime] [202]. I used the latter approach and implemented the lambda function in bash as calling the `xspec.sh` shell script is the simplest way to run the XSpec test suite. However, the lambda function could be re-written using virtually any programming language.
- Layers: the lambda function makes use of layers [layers] [203]. A layer is a zip file containing additional libraries and dependencies. In particular, I used layers for the XSpec runtime, the Saxon HE jar file, and bash [bash_layer] [202]. It is also possible to add a layer for Apache Ant and run the XSpec test suite via Ant which can speed up the test suite execution.
- Memory and Time Settings: these are configured in the settings for the lambda function. As running large XSpec test suites can be memory and time intensive, I recommend to adapt the values of memory and timeout accordingly. In my experience 512 MB memory and 2 minute timeout is a minimum threshold for running few XSpec tests.
- IAM configuration: the lambda function is granted permissions to interact with other AWS services like S3 and SNS via a IAM role [iam_role] [203]. It is a good security practice to allow access only to the relevant S3 buckets.
- Notification: I configured the lambda function to send email notifications via SNS (Figure 3 [201] shows an example of email notification with a failed test). It is also possible to configure SNS to send notifications to a chat messaging service like Slack and this may be more appropriate for large development teams.

Figure 3. Email notification



- CI Workflow: the lambda function is configured to stop as soon as a test fails: this is done in order to reduce the execution time and to lower costs. However, it is possible to adapt the lambda function to run the full test suite and report all the failing tests in the notification.
- Troubleshooting: debugging a lambda function can be challenging as the environment upon which it runs is stateless. Using CloudWatch to monitor the execution of the function and outputting the results of commands to the CloudWatch logs is extremely useful for troubleshooting.

4. Analysis and Discussion

In this section I analyze and discuss the major benefits, constraints and limitations of the serverless architecture for running the XSpec test suites.

4.1. Benefits of Serverless Architecture

Not having to provision and maintain servers and the software running on it is by far the greatest advantage of running a serverless architecture. Not only this can help reducing operational costs but it also frees up software engineering time that could be spent in more valuable tasks.

Another advantage is the scalability of a serverless environment once it is well architected in independent components. In fact, a serverless architecture is stateless and event-driven and can be easily scaled up and down by adjusting the resources and parameters of the single components. For example, memory allocation is a parameter in the lambda function; running out of storage space is not an issue since S3 provides virtually infinite storage space.

Finally, the serverless architecture is highly available since it relies on AWS managed services like API Gateway, Lambda, and S3: the architecture will go down only if and when AWS experiences an outage.

4.2. Memory and Time Execution Constraints

As described in the lambda configuration section, running XSpec test suites can be a memory and time intensive process and parameters for memory allocation and timeout need to be adjusted according to the number and type of XSpec tests.

AWS Lambda enables to run a function for up to 15 minutes, after which the function will automatically timeout. Therefore the XSpec test suite needs to run within this time constraint. This limitation is useful as it helps keeping the

test suite manageable and its execution fast enough to provide feedback to developers within a short time frame. As the test suite becomes more complex and more tests are added, it is recommended to break it down into different groups and assign a lambda function for each group of XSpec tests. This offers the advantage of running multiple lambda functions in parallel and keeps the total execution within the 15 minute time constraint.

4.3. Cost Optimization

Reducing operational costs is one of the major advantages of a serverless solution. However, costs in a cloud computing environment need to be closely monitored in order to avoid surprises.

AWS provides a billing console for monitoring and estimating costs. It also provides billing alarms when AWS costs go over a certain threshold. These are very useful tools for keeping the cost within the allocated budget.

Costs for AWS Lambda are based on usage and charged according to the number of requests and the duration [lambda_pricing] [203]. At the time of writing AWS provides a generous free tier of 1 million free requests and 400,000 GB-seconds of compute time per month. This makes a test suite based on a serverless architecture particularly attractive as it can be used with very little operational costs. However, increasing the memory and the timeout settings of the lambda function also increases the costs so these need to be factored in.

4.4. Vendor Lock-in

Amazon introduced AWS Lambda in 2014 and was the first cloud provider to offer and popularize this type of service in its cloud computing platform. However, nowadays other major cloud providers offer similar options for building serverless architecture: for example Microsoft with Azure Functions and Google with Google Cloud Functions. However, building serverless applications with a cloud provider inevitably comes with a degree of vendor lock-in as serverless services like AWS Lambda are proprietary and cannot be easily ported to another cloud provider. Therefore building serverless architecture may increase the depedency towards a single cloud provider.

5. Conclusion

In this paper I described an alternative approach for running XSpec test suites using a serverless architecture built on AWS. The high level configuration is explained in this paper and more details with step-by-step instructions and code examples are available on my GitHub account. I also analyzed and discussed benefits and constraints of a serverless solution.

I hope this can be helpful for development teams wishing to implement a CI workflow while reducing operational burden and costs. I would be interested in knowing and possibly helping anyone willing to implement this serverless workflow for running XSpec tests.

Bibliography

[api_gateway] Amazon Web Services, Inc.. *Amazon API Gateway*. https://aws.amazon.com/api-gateway . Accessed: 26 May 2019.

[bash_layer] GitHub. *Bash Lambda Layer*. https://github.com/gkrizek/bash-lambda-layer . Accessed: 26 May 2019.

[billing] Amazon Web Services, Inc.. *What Is AWS Billing and Cost Management?*. https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/billing-what-is.html . Accessed: 26 May 2019.

[cloudformation] Amazon Web Services, Inc.. *AWS CloudFormation*. https://aws.amazon.com/cloudformation . Accessed: 26 May 2019.

[cloudwatch] Amazon Web Services, Inc.. *Amazon CloudWatch*. https://aws.amazon.com/cloudwatch . Accessed: 26 May 2019.

[cncf] Cloud Native Computing Foundation. *CNCF WG-Serverless Whitepaper v1.0*. https://github.com/cncf/wg-serverless/raw/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf . Accessed: 26 May 2019.

[custom_runtime] Amazon Web Services, Inc.. *Custom AWS Lambda Runtimes*. https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html . Accessed: 26 May 2019.

[fowler] Martin Fowler. *Serverless Architectures*. https://martinfowler.com/articles/serverless.html . Accessed: 26 May 2019.

[git2s3] Amazon Web Services, Inc.. *Git Webhooks with AWS Services*. https://aws-quickstart.s3.amazonaws.com/quickstart-git2s3/doc/git-to-amazon-s3-using-webhooks.pdf . Accessed: 26 May 2019.

[iam] Amazon Web Services, Inc.. *AWS Identity and Access Management*. https://aws.amazon.com/iam . Accessed: 26 May 2019.

[iam_role] Amazon Web Services, Inc.. *IAM Roles*. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html . Accessed: 26 May 2019.

[kms] Amazon Web Services, Inc.. *AWS Key Management Service (KMS)*. https://aws.amazon.com/kms . Accessed: 26 May 2019.

[lambda] Amazon Web Services, Inc.. *What is AWS Lambda?*. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html . Accessed: 26 May 2019.

[lambda_pricing] Amazon Web Services, Inc.. *AWS Lambda pricing*. https://aws.amazon.com/lambda/pricing . Accessed: 26 May 2019.

[layers] Amazon Web Services, Inc.. *AWS Lambda Layers*. https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html . Accessed: 26 May 2019.

[markupuk2019] GitHub. *Markup UK 2019*. https://github.com/cirulls/markupuk2019 . Accessed: 26 May 2019.

[s3] Amazon Web Services, Inc.. *Amazon S3*. https://aws.amazon.com/s3 . Accessed: 26 May 2019.

[sns] Amazon Web Services, Inc.. *Amazon Simple Notification Service*. https://aws.amazon.com/sns . Accessed: 26 May 2019.

[xspec] GitHub. *XSpec*. https://github.com/xspec/xspec . Accessed: 26 May 2019.