Markup UK 2018 Proceedings

Markup UK 2018 Proceedings

Our Sponsors

EVOLVED BINARY

SAXONICA.com
XSLT AND XQUERY PROCESSING

®
XML Editor

Organisation Committee

Geert Bormans
Tomos Hillman
Ari Nordström
Andrew Sales
Bethan Tovey

Programme Committee

Achim Berndzen -
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
Mulberry Technologies
Deborah A. Lapeyre
Rebecca Shoob
Adam Retter
Jirka Kosek
Norman Walsh
...and our long-suffering partners

Sister Conferences

Markup UK 2018 Proceedings

by B. Tommie Usdin, Rob Walpole, Mark Dunn, Bert Willems, Tony Graham, David Maus, Makoto Murata, Hans-Jürgen Rennau, Hauke Brandes, Achim Berndzen, Philip Hodder, Michael Kay, Robin La Fontaine, and Steven Pemberton

Table of Contents

Shared Tag Sets as Social Constructs

B. Tommie Usdin, Mulberry Technologies, Inc.

1. Introduction

Let me start with a confession. These remarks are based solely on my observations and opinions. There is no science here, no controlled experiments, not even rigorously designed cultural anthropology. I would be delighted if someone did that work, but I have not. I have nothing but anecdotal evidence, and that is severely limited by the facts that I am only one person, am fluent in only one natural language, and have only been working with document markup for about 35 years (and databases for 10 before that), and work only in the world of prose documents. If you tell me that you have different experiences, I will not only not tell you that you are not wrong, I will ask you to tell me about it. There isn't one way this always works; there isn't one pattern of interactions that works for everyone; there isn't one pattern in which all markup communities interact. These are my opinions, not facts.

2. From the Mouth of a Novice

Not long ago I had an interesting conversation with a young man who had just written his first XML document-processing application (of course, he wrote his own document vocabulary). He then showed it to some of the senior people in the organization where he works, and was shocked and disappointed at the responses he got. He had clearly done a lot of reading, worked hard, and build something to impress the powers that be. He wanted recognition, praise, and perhaps additional responsibilities. They laughed at him.

He is the son of a friend of a friend, so he felt safe asking me what had gone wrong. The first thing I told him is that the big thing he had done right was to dive in and learn about something he found interesting but knew nothing about. I thought that was laudable, and that spirit would serve him well in the future.

Now, let's look at some of the problems with his XML vocabulary. He said that they laughed at him because:

- What it Looks Like: "They told me it was bad XML because there are tags for what things look like and what I want the app to do as well as some for what the stuff means. They said good XML is about meaning, not about look and feel and that, when tagging regular documents, the formatting should be separate from the content."

- Types of Markup: "They talked about generic markup and structural markup and containers and procedural markup (which they said was 'deprecated' - a word I did not know but that sounds pretty snooty to me)."

- Mixed Metaphors: "They told me I should not write my own vocabulary until I had made sure one of the 'standard' ones would not work for me. Hey, I started with DocBook, but my project is really special, and DocBook did not define all I needed. Then my librarian friend told me that the reference model in JATS is better than the one in DocBook, so I copied it in. Why can't I do that?"

- Natural/Intuitive Tagging: "I also made my language a lot more natural by using tags from HTML whenever I could. You know, so people would know what the tags meant."

And he said: "I read the XML recommendation":

- "I didn't see anything in the XML recommendation that says anything about avoiding look and feel, and besides, XSL-FO documents are XML and they are all about what the page should look like."

- "And the XML recommendation never mentioned: procedural markup, descriptive markup, generic markup, or structural markup. If this stuff is so important to making good XML why doesn't the XML specification talk about it?"

- "If it's not in the rec, how is a person supposed to know it's important and how do I learn about it?"

3. The Bare Bones XML specification

Well, in a way, he is right. The XML specification does not address separation of content from format, or most other design decisions, and XSL-FO *is* in XML document syntax and is about page formatting. Yes, but.

XML was written as the 'good parts version' of SGML. It is a short document that concentrates on document grammar and well-formedness; DTD grammar and syntax, validity constraints, entities, processors, and the like. It assumes that you want to know *how* to define a markup vocabulary and says nothing about:

- Why tag information

- Separating formatting/look-and-feel/behavior from content

- Procedural markup versus descriptive markup

- Generic markup

- Structural markup

- Semantics

- Best (or even recommended) practice for modeling vocabularies

- Data versus documents

- Architectural forms and reuse

- Separation of concerns

- What should or should not be expressed in a grammar

- Element versus attributes

Document-based XML is assumed, by large numbers of its practitioners, to include:

- Generic Markup To many of us, the idea of XML incorporates "Generic Markup" that is, the art of identifying information by what it is, not by what it should look like at any moment or how it will be used in any one place. When making up tags for "regular documents" (what ever that means), it is generally considered better if you use generic markup rather than format-based markup or behavioral, procedural markup. Much, but not all, XML benefits from an underlying skeleton of generic markup.

- Separating format from content The principle of separating content from format (what it is not what it looks like) is another formulation of the same general idea, although it goes a bit further in that it implies that there can be more than one display or use for the same content. This is, in fact, one of the main virtues of XML as it is generally used. We can take the same tagged text, extract from it, superset it, repeat anything we need, delete what we want, and each of us make it look-feel-behave in an entirely different way.

- Separation of concerns Some people think there should be a schema with a widely used grammar for a document type, with perhaps several sets of Schematron rules to enforce more local business rules that are also imposed on those documents, other validations as necessary, and then we can each use a different stylesheet to make it look like we want. There are long discussions of what should be imposed based on a grammar, and what should not.

But the XML Recommendation mentions none of that.

4. Advantages of Shared Vocabularies

And then my friend was told to use a shared (common, public, standard) vocabulary.

XML grew out of SGML. SGML grew out of an effort to make a universal tag set for typesetting. (Yes, that is an oversimplification, but it is based on truth.) We couldn't make a universal tag set because we figured out that this would be making a list of everything that ever mattered to anyone; a task that could not be finished. So instead a syntax for grow-your-own-markup-vocabulary was designed. Shortly after that, people started getting together in groups to make markup vocabularies for specific communities or uses. Some of those have been widely adopted (as recommendations and specifications, and even standards) and are at the center of communities of users. Some of the big winners in this game: DITA, HTML, NIEM, JATS, TEI, UBL, HL7, DocBook, NISO STS, BITS,...

There are also thousands of other vocabularies that are not as widely known, discussed, or adopted. Are they failures? Some are. I have written a few vocabularies that were intended to be widely adopted that have no users and never did. Those are failures. But there are others that are at the heart of a dozen or so businesses or activities. Are they failures? Absolutely not. They are enabling their users to create, validate, interchange, and use their content. We don't need mega-success to have success. You can roll your own successfully. So what do shared vocabularies buy us?

The enthusiasm for some shared markup vocabularies is both charming and (sometimes) absurd. Useful and destructive. The impulse of people with similar concerns and needs to get together to create shared markup vocabularies and shared infrastructure to create, manage, and use documents tagged according to this shared vocabulary is admirable, and is the reason there are so many useful markup applications in use. If every user needed to define their own tag set, document it so it was used consistently among various document creators and over time, and write all of the tools needed to create, validate, edit, display, search, and archive their documents, few would make the investment and fewer would stick with it after the first or second bump in the road. Further, it would be virtually impossible to interchange marked-up documents; each organization that wanted to receive documents would need to develop a custom transformation for each document type they wanted to ingest into their system.

While a world in which each user defined their own optimal tags would be within the definitions of the XML specification, and some people reading the specification might assume that is what was intended, that world would be very inefficient, tag-set heavy, and isolated. Fortunately for all of us, except perhaps those who revel in vocabulary creation and coding transformations – that is not the world we live in.

Markup vocabularies also have traction in communities, partly because using a known vocabulary means that knowledgeable help with be available. If you are a scholar doing literary analysis you probably want to spend your time and energy on the analysis and recording of your documents, not on your infrastructure. You would be well advised to use TEI markup (Text Encoding Initiative academic tag set) partly because it was designed to do what you are doing, but *maybe especially* because tools, discussion lists, tutorials, and friendly advice are easily available if you do. And because your colleagues, who will review your funding proposals, journal articles, and your promotion and tenure applications, expect you to.

4.1. Of course All is not sweetness and light

Sometimes the enthusiasm for a vocabulary gets absurd. I recently over heard a conversation at a markup-related meeting between two people who had not previously met. One introduced himself as new to XML and there to learn. The other said "You need to learn about MYVOCABULARY. Don't waste your time on anything else, MYVOCABULARY is the only smart thing to do in 2018." Note: this person was recommending a specific vocabulary to a new user without knowing anything about the new user's data, requirements, or environment. This is stupid. This happens way too often. I have heard a software sales person tell a potential customer that if they didn't select the markup vocabulary this person's software best supported, the sales person would call the potential clients manager and tell the manager that the would-be client was incompetent. Fortunately, the object of this bullying was not intimidated, and made a decision that was appropriate for the organization and document collection, not for the salesperson.

I have seen it go in the other direction, too. A user with a need saw an application working at another user site, really liked how it was working, and asked about it. We are using this and that tool, our XML is encoded using this vocabulary, and we are getting this performance. Wow! So, the would-be user called those vendors, learned that those tools could be customized to work with almost any XML vocabulary but out of the box (or with minimal customization) worked with the one they had seen demonstrated. So, this user had a customization of that shared vocabulary made to meet their needs, minimize the cost of customizing those impressive tools, and get up and running as quickly and economically as possible. Was it the best vocabulary choice…?

4.2. Mixing Vocabularies (by cut and paste not namespaces)

Mixing vocabularies. My young friend did not realize that you lose a lot of the value of using a public vocabulary when you pick and choose snippets from one vocabulary and another and then mix them together. You can make a real mess by copying and pasting declarations from one schema into another totally disregarding those pesky namespaces.

The reasons that people got together to create those public vocabularies is so that they can easily and conveniently interchange documents, so they can share in the costs of customizing tools and other document-related infrastructure, and so that they can move from collection to collection and organization to organization without having to start all over learning an unfamiliar application. Also, because habits take a while to develop and once you have them it is easier to work in an environment in which those habits work than to break old habits and form new ones.

By mixing and matching from among several public vocabularies you are walking away from all of the traditional advantages of using them. You will create documents that no one else will be prepared to receive or process, so you have no interchange. You will not be able to use tool customizations or stylesheets that have been developed by any of the use communities. You will not be able to take advantage of the economies provided by using vendors that already know your requirements, nor will you have the built in quality checks that come of vendors knowing your vocabulary.

4.3. Natural/Intuitive Tagging (Influences of Shared Vocabularies on Thinking)

Sorry, my young friend, there is nothing more "natural" about HTML tags than any other tags! Intuitive markup is like intuitive user interfaces, a lovely but very fragile environment-based fiction.

Familiarity with one markup vocabulary often influences others. In the last 25 years or so, HTML has become such a well-known vocabulary that many users don't think of it as a set of tags that someone made up and that have grown over time. Many users think of the HTML tags as "natural" in some fashion. So, they think it is "natural" to use <p> for paragraphs and <li> for list items. Even odder, they think it is "natural" to have different wrapper tags for numbered and unnumbered lists but to distinguish between numbered and lettered lists with an attribute. I'm not saying that this doesn't work; clearly it does. But there is nothing more "natural" about this than, for example, having a single list wrapper element and distinguishing between unordered, numbered, and lettered display with an attribute. Users have objected to using unfamiliar tags for familiar structures; we already know HTML, we should use HTML tags when possible.

More insidious, we already know the logical model that underlies HTML so well that we think of it as the "natural" model of text. HTML does not allow lists inside paragraphs. Therefore, even in the case where a list occurs inside a sentence, which continues after the end of the list, where the list is obviously inside the paragraph, we should encode that as two paragraphs with a list between them, because HTML would have to code them that way. I have been astonished by users who were shocked by the suggestion that in their content they might allow lists to occur inside paragraphs instead of between them!

Of course, we get push-back in the other direction, too. A comment on JATS recently suggested that the "rather old fashioned HTML style (<i>,<b>), e.g. <bold>, <italic>" tags be replaced by tags such as <emph> or <hi> "that at least try to encode the 'meaning' of the tagged content". This, in my opinion, is simply a stylistic suggestion: <i>, <italic> and <emph type="italic"> carry exactly the same amount of semantic information! The suggestion was to use a more familiar or more comfortable tag set, not to increase the semantic richness of the vocabulary.

5. Shared Wisdom: the culture and practice of XML

So how do we learn XML in general and our shared tag set in specific? That's harder. People learn by watching others. It is not unlike the old apprentice system, and you can hope you are apprenticed to a master, not a dud.

Learning a tag set and good XML practice is sort of a community values thing, transmitted over discussion lists, at conferences, and through examples. Some, but far from all, introductions to XML and principles of XML design address generic markup, separation of content from formatting, structural markup versus and procedural markup. Some go even further, and talk about separation of concerns at several levels, for example separating the grammar of a document type from local or temporal business rules. But large tag sets are very diverse.

5.1. Tag Set Cultures

Shared tag sets are much more than lists of codes, they are powerful and dynamic social constructs that influence and are influenced by the world.

There is a basic culture surrounding the "practice of markup", especially as practiced in pervasive, shared XML vocabularies. Each culture, formed in the confluence of people creating, using, and interchanging documents, is both large and quite separate from the standard, specification , or recommendation that defines the tag set. The culture defines the customs and rules about how to interact with and use the tag set. This culture includes mentors and consultants, documentation, webinars, discussion lists, courses, books, conferences, recommended practices and examples, as well as shared tools. Shared tools may be available for both tag set and document creation and editing, document processing including format and display, validation and rules checking, and much more.

The needs of those groups, and the changing needs of the various members who have joined or left these activities, have shaped the tag sets. Things the membership, or the sponsors, or the most vocal/powerful members, think are important are enabled and/or emphasized. (For example, we are seeing more and more tag sets providing the ability to enhance documents with Accessibility information as access becomes more important in the world in general.) How the movers and shakers think influences and molds the tag set.

Conversely, shared tag sets, and the assumptions that underlie them, shape the world view of their users. Before HTML it was common to build tagging structures that allowed paragraphs to contain lists; these days that is considered surprising.

Many of us have bought into the OHCO (Ordered Hierarchy of Content Objects)Renear, Mylonas, and Durand 1996 [6] or "all the world is a tree" model, because XML does those things well. Before SGML and XML it was not unusual to discuss how to identify, markup, and work with the overlapping structures in texts; now it is often considered an "edge case" and thus ignored. To many in the XML world, overlapping structures are simply unthinkable; we have tree-shaped scars in our minds from working with tree-shaped structures. This does not mean that there aren't significant applications that have, and need, overlapping structures.

I have heard "serious XMLers who work with serious documents" dismiss the "overlap fanciers" as airy-fairy academics who don't need to get "real work" done. It is fun to remind them that standards, aircraft documentation, and military specifications all require strict and accurate change tracking, and that changes often overlap the initial structure of the document.

5.2. Different use cases make for different cultures

Each tag set community has different focuses, practices, and values based on their use cases. For example:

· DITA and DocBook are typically concerned with content creators creating content: authoring (and reuse) by people.

- JATS and BITS were designed as conversion targets; they typically assume that content already exists (probably created in Microsoft Word or legacy in PDF) and needs to be marked up for interchange or archival purposes. (There is an authoring version of JATS, but it is very rarely used.)

- TEI typically deals with previously-created or historical content, marked up by subject matter specialists who are content analyzers and annotators.

Such differences in the who and why lead to significant changes in the mindset of the designers and developers, in the structure of the tag sets, in the nature of use, in the documentation, and in the tools created and shared.

5.3. Community coercion: All is still not sweetness and light

The communities around various tag sets may pressure users to encode information they do not need in documents because the community expects it: TEI users may feel pressure to provide more metadata, JATS users to provide richly encoded citations, HTML users to use more generic tags in place of visually descriptive tags.

As an example, once you have drunk the TEI Kool-Aid, the TEI community may pressure you to encode your documents in ways that are useful to the full community, even if some of that encoding is of no value to you and your project – because someone else may want to reuse your documents for another project at some indefinite time in the future.

This "do it for the good of the community" is not unique to the TEI. "Do a bit extra for the community" is the energy that is expected to (that does?) power the semantic web. It is also the hope behind groups like JATS4R. JATS is the current most popular Journal Article Tag Suite and JATS4R (JATS for Reuse) is a group of JATS users who believe that the JATS vocabularies and guidelines are too loose; there is too much variation in legal JATS documents to make it convenient to reuse them, particularly for machine reuse such as data-mining. So, JATS4R is developing and publishing a set of guidelines and some Schematron rules to help the creators of JATS documents make more similar use of JATS. For example, JATS provides many ways to encode author names, author affiliations, and the relationship between authors and their affiliations. JATS4R recommends a smaller number of ways to do this (ideally just one).

5.4. But ...

But in spite of any potential downsides, such a culture surrounding a tag set is essential to the effectiveness of the tag set, possibly more important than the shared elements, schemas, or "semantics".

And there is shared culture in the XML space. Self-taught XMLers have a very difficult time making good XML; designing what a friend of mine calls "colloquial XML applications". Reading specifications is enough to enable the creation of legal, and probably functional, markup applications, and might enable a genius to create something fresh, brilliant, and better than anything we are doing now. Most of us are not geniuses, most of us are at best capable with occasional moments in which we shine. And most of us benefit greatly from participating in the culture of the communities in which we work.

6. Conclusion

Markup culture is time-saving, energy saving, and application enabling. It is transmitted through books, mailing lists and online discussions, classes and tutorials, and conferences.

MarkupUK is valuable because it is a place for us to learn from each other. To learn not what just someone else is doing but how they thought about it, how they approached their tasks, how they selected approaches, how they evaluated results, and how they describe their successes and failures.

Bibliography

[Renear, Mylonas, and Durand 1996] Renear, Allen H., Elli Mylonas, and David G. Durand. " Refining our notion of what text really is: the problem of overlapping hierarchies. " *Research in humanities computing* , ed. Nancy Ide and Susan Hockey. Oxford: Oxford University Press. 1996. http://cds.library.brown.edu/resources/stg/monographs/ohco.html [http://cds.library.brown.edu/resources/stg/monographs/ohco.html]

How to Make a Flying Start with S1000D – Lessons Learned at the Airport

Rob Walpole, Flatirons Solutions

Abstract

S1000D is a specification for tech pubs which has been broadly adopted in the aerospace industry. It provides XML schemas for common doc types and defines how these documents should be structured in a common source database. S1000D is replacing the ATA SGML format as the primary spec for OEM aircraft manuals and is used for new aircraft such as the Boeing 787 and Airbus A350. At the same time, mobile devices are playing an increasing role in aircraft maintenance, allowing engineers to access technical information at the point of need.

Airlines are pressured to keep fares low but still maintain the highest safety standards. A single source for engineering content is one way to reduce overheads. It follows that other documents for engineers should also be S1000D to allow common tools for managing and rendering content.

This paper discusses implementing S1000D for a major UK airline's engineering procedures. While OEM manuals cover much information needed by engineers, airlines still need their own procedures for topics like health and safety, planning and quality control. In fact, to cover all aspects of keeping aircraft functioning safely and comfortably for passengers.

S1000D can be daunting for the inexperienced though. The version 4.1 spec is over 3500 pages long and not an easy read. There is a shortage of S1000D information in the public domain, perhaps because it is still quite new, the majority of aircraft manuals are still in SGML and the industry is understandably cautious about change.

Of course, you could skip the spec and start creating some S1000D schema compliant XML, only referring to it when you hit a snag. This is a very bad idea. By this time, you will likely have created a number of problems for yourself which will be hard to unwind. This paper will explain some of the design decisions you need to make before going anywhere near a keyboard and hopefully save you from some of the pain that you will inevitably experience.

Note

Unless otherwise stated, all mention of S1000D in this paper is specifically referring to issue 4.1 of the specification.

1. Introduction to S1000D

1.1. What is S1000D?

S1000D describes itself variously as an "international specification for technical publication using a common source database"[[1] [18]] and "an international specification for the procurement and production of technical publications"[[2] [18]]. Essentially it is an XML specification designed to standardise data exchange and enable both producers and consumers to create, manage and exchange content without being tied to any one proprietary solution. It is predominantly used for defence and civil aviation purposes and increasingly in areas such as construction, shipping and the automotive industry. The specification itself notes that there is nothing to stop it being used for non-technical publications as well[2] [18].

1.2. History

The S1000D concept originated in the early 1980s and was initially developed by the AeroSpace and Defence Industries Association of Europe [http://www.asd-europe.org/] (ASD) for military use. At the time there were various national military specifications for documentation. As computers were becoming more prevalent there was a drive to develop computer-based methods of document management. To avoid having too many different, complex and expensive systems, it was apparent there was a need to standardise the documentation methodology.

At the time most civil aviation projects were documented using the ATA 100 specification, developed by the Air Transport Association of America (now known as Airlines for America [http://airlines.org/]) in the United States and first published in 1956. Originally this specification was just a document numbering system but has developed over time into the current ATA iSpec 2200 specification which also covers content, structure and exchange of engineering and maintenance information from manufacturer to operator [[3] [18]]

ASD decided to set up the Documentation Working Group (DWG) to try to harmonise civil and military documentation and to use ATA 100 as a base. The group realised that creating a harmonised specification would have many advantages including[[2] [18]]:

- cost saving in information generation - avoidance of duplications

- more economic support planning

- cheaper deliverable publications

- uniformity of standard for participants in the project

- standard format for data exchange to exploit future developments

- enhanced interoperability

Early versions of S1000D provided a set of Document Type Definitions (DTDs) for producing Standard Generalized Markup Language (SGML) but gradually it became and remains an Extensible Markup Language (XML) specification defined in XML Schemas.

Since 2003 ASD has been working with the Aerospace Industries Association of America [https://www.aia-aerospace.org/] (AIA) and, since 2005, with the ATA e-Business Program [http://www.ataebiz.org/Pages/default.aspx] as well, to harmonise US and European technical publication guidance. These three organisations are now working together to develop, maintain and promote S1000D internationally.

1.3. Principal Concepts

The S1000D specifications defines a number of principal concepts which allow the creation, management and exchange of technical publications. Perhaps the most important concept behind S1000D is that of modularisation; the idea that a document can be broken down into individual segments known as Data Modules that can then be reused across multiple

publications. This concept allows content to be updated in one location and for that change to then appear in multiple publications.

1.3.1. Data Modules

A data module represents the smallest self-contained information unit within a technical publication. Exactly what constitutes a data module is for the author to decide and it will depend on the size of the publication and how much content is suitable for reuse. A data module could represent an entire document, if it is only a few pages and the content is only suitable for re-use in its entirety; or it could represent a chapter, section, sub-section, paragraph or something as small as a list, if that list could be re-used in multiple documents. The data module consist of a single XML file with a root element <dmodule> and two main sections: the identification and status section <identAndStatusSection> and the content section.

```
<dmodule>
    <identAndStatusSection>
        <dmAddress>
        ...
        </dmAddress>
        <dmStatus>
        ...
        </dmStatus>
    </identAndStatusSection>
    <content>
    ...
    </content>
</dmodule>
```

The identification and status section contains the metadata and management information and is further broken down into the data module address element <dmAddress> which contains identification information such as numbering, revision and title, and the data module status element <dmStatus> which contains information such as providence, security classification and applicable business rules.

The content section of the data module contains the text and illustrations displayed to the user. The structure of this section varies depending on the type of publication and the data module schema being used. The available schemas (version 4.1) are shown in the following table:

Table 1. S1000D Data Module Schemas

Data module type	Schema
Applicability cross-reference table (ACT)	appliccrossreftable.xsd
Business rules exchange information (BREX)	brex.xsd
Checklist information	checklist.xsd
Common information repository (CIR)	comrep.xsd
Conditions cross-reference table (CCT)	condcrossreftable.xsd
Container information	container.xsd
Crew/operator information	crew.xsd
Descriptive information	descript.xsd
Fault isolation information	fault.xsd
Front matter	frontmatter.xsd
Learning information	learning.xsd
Maintenance planning information	schedul.xsd
Parts information	ipd.xsd
Procedural information	proced.xsd
Process module information	process.xsd
Products cross-reference table	prdcrossreftable.xsd
SCO (Shareable content objects) information	scocontent.xsd
Service bulletins	sb.xsd
Wiring data	wrngdata.xsd
Wiring data description information	wrngflds.xsd

1.3.2. Publication Modules

Publication modules define the content and structure of a publication. Content can be packaged and delivered as one or more publication modules, each of which contain a series of references to data modules or other publication modules. Every publication has a single top level publication module. As with data modules, the degree to which publications are split and broken down by sections is a matter for authors to decide. A publication module is represented by a single XML file with a <pm> root element and the same two main sections as a data module but with the content section containing one or more publication module entries.

```
<pm>
    <identAndStatusSection>
        <pmAddress>
        ...
        </pmAddress>
        <pmStatus>
        ...
        </pmStatus>
    </identAndStatusSection>
    <content>
        <pmEntry>
            <pmRef>
            ...
            </pmRef>
        </pmEntry>
        <pmEntry>
            <dmRef>
            ...
            </dmRef>
        </pmEntry>
    </content>
</pm>
```

A publication module entry <pmEntry> can contain references to other publication modules <pmRef>, data modules <dmRef> and also contain other <pmEntry> elements if the entry exists only to give structure to the publication. The <pmEntry> elements can be thought of as providing the table of contents, so there may be a heading which contains only other sub-headings for example. A <pmEntry> can also have a title <pmEntryTitle> which is seen by the viewer. If no title is provided then alternatively the reference number can be shown to the viewer or the title could be derived from the target document.

1.3.3. Common Source Database

The S1000D specification defines the Common Source Database (CSDB) as "an information store and management tool for all objects required to produce the technical publications within projects"[[4] [18]]. In practice that means that it is the repository for all data modules and publication modules plus any graphics or other multimedia used. It must also contain supporting management files such as Data Dispatch Notes (DDN) and Data Management Lists (DML). The specification explicitly does not define how a CSDB should be implemented.

The most important concept behind the CSDB is that it is a single source for content and should not contain any duplicated data. Data modules should be sufficiently granular in their content to allow re-use in multiple outputs. The word 'module' in 'data module' and 'publication module' has been carefully chosen. Publications can be built up of many modules. There is nothing to stop different publications re-using the same data module, in fact this is positively encouraged. It means authors only need to update content in one location and multiple publications will contain the revised content.

2. Introduction to the Knowledge Warehouse Project

Knowledge Warehouse was the name of a legacy SAP document management system used by a major UK airline to manage engineering procedures, work instructions, user guides and other content created in house for use by aircraft engineers. Much of the content was in Microsoft Word format but it also contained Excel spreadsheets and PDF files.

The airline had already procured an ATA/S1000D content management system (Flatirons Solutions CORENA Suite [https://www.flatironssolutions.com/]) that would enable their engineers to access OEM aircraft maintenance manuals

via tablet computer at the point of need. It therefore made sense to migrate the Knowledge Warehouse content to the same platform so that engineers could also view this content on their tablets using the same applications. Prior to this, engineers had to access on-line content via computer terminals in the hangar. This would mean walking from the aircraft to the terminal between each job, printing the relevant content and then returning to the aircraft. While no huge distances were involved, any time and motion study would show that being able to access the content from anywhere in the hangar via tablet has the potential to bring significant time and cost savings.

In order to manage these documents on the same platform as the OEM manuals it was decided to convert them into, and maintain them as, S1000D. This would have the added benefit of turning them into structured XML documents which would allow powerful searches to be performed using XQuery and multiple representations to be generated for different devices using XSLT. It would also mean that the content could be managed using the same tool-set as the OEM manuals.

3. Migration Planning

Migrating publications to S1000D is essentially a matter of creating data modules for content and publication modules for structure. A decision was taken to reuse the navigation structure in the legacy system to create the publication module structure and to convert each document into a single data module. The only data module schema that was found to be suitable for all of the very varied content was the descriptive schema which allows a reasonably free content layout. Although some of the documents were procedures and might have benefited from the procedural schema, this schema was found to be too strict for the airline's needs.

Table 2. A word of advice..

> This was perhaps the first point in the project where we came up against a fundamental fact of S1000D, namely that its development has been heavily influenced by the needs of product manufacturers. Because of this, tailoring S1000D to fit wider applications can be a challenge. Aspects that have a clear purpose when talking about a specific product are hard to make sense of when talking in more general terms. For example, the procedural schema insists on a preliminary requirements section. This might make sense when dismantling a specific component from an aircraft, where the component first has to be removed and certain tools will need to be on hand, but when talking about a more general procedure, for example cleaning up a spillage, then that probably depends on the type of spillage and you might want the freedom to explain that.

3.1. Data Modules

The legacy system was able to produce an HTML [https://www.w3.org/TR/html/] rendering of the Word documents it stored. By converting this HTML to XHTML [https://www.w3.org/TR/html/] (using the html2xhtml [http://www.it.uc3m.es/jaf/html2xhtml/] command line tool) and then transforming the XHTML to an S1000D data module using XSLT [https://www.w3.org/TR/xslt/all/], it was possible to obtain a rough and ready but valid S1000D document which could be manually enhanced at a later date.

For PDF documents it was decided to create the data module using the text extracted from the PDF using the iText [https://itextpdf.com/] library. Although it was not possible to preserve any of the document's internal structure in this way, the PDF could also be stored in the CSDB as a binary resource. As an interim solution the users would see the PDF document when accessing the data module until the data module had been enhanced to resemble the PDF, at which point it could be removed.

While this approach worked well for the vast majority of documents there were a small number of very large documents that were later discovered to be too big to manage efficiently in the CSDB. Working with them in an XML editor was slow or even impossible on certain low specification workstations and rendering them for viewing could also be problematic.

Table 3. A word of advice..

> With the benefit of hindsight, a review of very large documents should have done before they were converted to S1000D. Breaking these documents up in their original format would have probably been easier than breaking them up after they had been converted to S1000D and loaded into the CSDB.

3.1.1. The Data Module Code

Before it was possible to generate any S1000D documents at all, an identification numbering system, or data module code, had to be devised as this information is stored internally in the data module, in the <indentAndStatusSection> mentioned earlier, and also forms part of the data module file name. The data module code is built up of a number of sections which are described in detail below.

Table 4. A word of advice..

> Once again S1000D clearly shows its roots in component manufacturing. Codes such as model identification, system difference, subsystem and assembly make a lot of sense when talking about the components of an aircraft engine but not so much in other contexts such as general procedural documents and it can be hard to make meaningful sense of them.

3.1.2. Model Identification Code

The Model Identification Code (MIC) is used to identify an individual product or project. To be valid it has to be between 2 and 14 characters long which can include only numbers and upper-case letters. According to the specification, projects must apply to the North Atlantic Treaty Organisation (NATO) Support and Procurement Agency (NSPA) to register an MIC. In practice however, this is more relevant to equipment suppliers and if the data is not going to be exported to other organisations then it should be sufficient to think up a code which is (a) unlikely to be used by anyone else, perhaps by including an organisation name, and (b) which is not one of those already registered. The registered codes can be seen at http://www.nspa.nato.int/en/organization/Logistics/LogServ/asds2000m.htm

During the Knowledge Warehouse migration project, the model identification code went through several iterations of change. Initially a very project specific name was chosen but it was soon realised that the project name would become meaningless over time. A thought experiment was carried out to envisage what content might be managed as S1000D in the future. This led to a very high-level name which would cover any content from anywhere within, not only the airline, but its parent company as well. This idea was soon dropped when it was realised that the rest of the numbering system would be exhausted long before it could manage that many documents. In the end a name which was airline specific and signified internal company documents was chosen. If other types of documents from the same airline or documents from other associated organisations were placed in the repository then new model identification codes would be chosen for them.

3.1.3. System Difference Code

The System Difference Code (SDC) is intended to identify different versions of a system. To be valid it must be between one and four characters long which can only include numbers and upper-case letters. Once the model identification code had been selected, choosing a system difference code was found to be relatively easy. As they MIC identified internal airline documents it was natural to use airline divisions for the SDC. As the migration project only covered the engineering part of the business a code that signified this was chosen.

3.1.4. Standard Numbering System

The Standard Numbering System (SNS) consists of groups of characters (numbers or upper-case letters) separated by dashes. The first group of two or three characters is the system code, the second group, rather confusingly, has one character for the sub-system and another for the sub-sub-system code and the final group of two or four characters is the assembly code, e.g. XX-XX-XX or XX-XX-XXXX

Once again component manufacturing shows its strong influence on the specification. One way to make sense of this is to think of the three groups in terms of chapter, section and subject. The Knowledge Warehouse content already had a number of top level headings which could be thought of as chapters, which themselves were divided into sections, so this seemed at first a straightforward mapping. It also highlighted an important point, which is the relationship between the content in the repository and what is seen by the viewer. While S1000D advocates a modular approach, meaning content can be re-used anywhere, it is also important to consider that authors need to be able to locate the content quickly in the CSDB. Having a numbering system that means something is one way to achieve this. If an author knows they want to edit a procedure and all of the procedures are in chapter 1 which has a SNS starting 01 then this makes locating the content in the repository much easier than if a randomly allocated number is used. It was also discovered that having to consider a new numbering system prompted a rethink of how the airline's engineering content was structured. For example, several chapters contained a section on health and safety. While some of this content could be re-used in a modular way, some was unique to the context of the chapter. It was decided to use the same section number to signify health and safety in each chapter so that authors knew immediately where to find health and safety content, as shown below:

01-02-XXXX – Health and safety section in chapter 1

02-02-XXXX – Health and safety section in chapter 2

Table 5. A word of advice..

> With hindsight, perhaps it would've been even better to put all Health and Safety content under a common code, maybe using HS as the system identifier and forgetting about the chapter context all together. In truth, there are many different and equally valid ways to organise content in S1000D and in the end it is for those who will be using the system to decide what works for them.

Another important factor to consider is allowing space for the content to grow. For this reason it was decided quickly to use 4 characters for the assembly (or subject) code so that no artificial constraints were created early on in the project.

Similar restrictions exist on the total number of chapters and sections available, depending on whether it is decided to use numbers only or letters as well. Whatever system is chosen it is important to ensure that there is plenty of growing room at every level.

3.1.5. Other Codes

3.1.5.1. Disassembly Code and Disassembly Code Variant

The Disassembly Code (DC) and Disassembly Code Variant (DCV) are two parts of the data module code which are very hard to make sense of outside of a component manufacturing or maintenance context. They relate to the idea that a dismantled part is made up of a series of other parts. They are however compulsory parts of the data module code so, for the purposes of the Knowledge Warehouse project, they were left as zeros.

3.1.5.2. Information Code and Information Code Variant

The Information Code (IC) is used to identify the type of information in the S1000D document. The specification defines a number of generic information codes and it is also possible to define custom codes, specific to a project. For simplicity it was decided to use one generic code for all Knowledge Warehouse content: "040" meaning "description" which fits with the use of the descriptive schema. The Information Code Variant (ICV) is intended to signify variations of the activity covered by the information code. The default value is the character A.

3.1.5.3. Item Location Code

The Item Location Code (ILC) is another very component specific code which had no meaningful use in the context of Knowledge Warehouse. As a value is mandatory the character D was chosen meaning all locations apply.

3.1.6. Putting It All Together

The complete data module code is put together as shown in the example table below.

Table 6. Data Module Code

MIC	SDC	SNS	DC + DCV	IC + ICV	ICL
AIRLINEDOCS	ENG	01-01-0001	00000	040A	D

When used in file names or URIs the data module code is preceded by "DMC" and the elements joined together with dashes, for example:

DMC-AIRLINEDOCS-ENG-01-01-0001-00000-040A-D.xml

3.2. Data Module Title

A data module title <dmTitle> is made up of 2 elements, the technical name <techName> which is mandatory, and the information name <infoName> which is optional.

```
<dmTitle>
    <techName>Headset</techName>
    <infoName>Install procedure</infoName>
</dmTitle>
```

If the information name is present then it is normally appended to the technical name when displayed. So the example above would be displayed as:

Headset – Install procedure

For the Knowledge Warehouse content being migrated, document titles were already well established and in fact most documents already had two titles, the name and the document reference number. For example: *Airplane Health Management and Airman Systems – EN-WD-3-5*. While an argument could be made that the historic reference number was being replaced by the data module code, many users had memorised the reference number and would search for this

rather than the document title. It was therefore a straightforward decision to map the document title to the <techName> and the document number, where available, to the optional <infoName>.

Table 7. A word of advice..

> One of the most important points re-enforced by this project is not to alienate your users. S1000D may be a totally new concept to them but frankly they are unlikely to be impressed unless they can find the content they need quickly. Listening carefully to the users and understanding how they want to use the system is essential for a successful S1000D migration. A certain degree of adapting is going to be inevitable for them but they should not be forced into new ways of doing things if it can be avoided.

4. Publication Modules

The legacy Knowledge Warehouse system had seven top-level navigation headings, so in order to provide the same top-level navigation in S1000D, it made sense to create a publication module for each heading. Creating the publication modules seemed straightforward as the legacy navigation structure was available as HTML and, by converting this first to XHTML, it was possible to transform the navigation structure into S1000D XML publication modules using XSLT.

Table 8. A word of advice..

> Although this was a relatively straightforward task, with the benefit of hindsight copying the existing navigation exactly was not the best decision. It was later discovered that some of the publication modules were so large that it was almost impossible to work with them in an editor. These publication modules had to be split later into multiple smaller publication modules which were then referenced from the top-level publication module, making it a much smaller and more manageable file. Reviewing the number of entries at each level in the structure before the migration began, and restructuring where prudent, would've avoided this extra work later.

In order to match the entry in the publication module with the correct data module, a title lookup was used. It soon became apparent that in some cases the lookup was unable to resolve the correct data module as multiple data modules had the same title. Further investigation showed that the reason for this was that the legacy system did not support content re-use and therefore the same document had been copied into multiple locations in the structure as it was needed. On examination it was found that the documents were not always identical, some having been edited and this created inconsistency in the content. This is exactly the kind of situation that S1000D is designed to prevent through module reuse. To resolve the problem a manual effort was needed by authors to merge the content and remove the duplicates prior to running the migration.

4.1. The Publication Module Code

4.1.1. Model Identification Code

The Model Identification Code (MIC) used would normally be the same as that chosen for the Data Modules and this was the case for Knowledge Warehouse.

4.1.2. Issuing Authority

The Issuing Authority is the organisation that produced the publication. This five-character code can be made up of numbers and upper-case letters. The specification recommends using a Commercial and Government Entity (CAGE) code [[5] [18]] for the issuing authority, which happens to fit the criteria exactly. CAGE codes are widely used and many aerospace organisations, including airlines, are already using them, as was the case in the Knowledge Warehouse project. If in doubt there are various CAGE code search engines available online such as https://www.cagecode.info/.

4.1.3. Publication Number

The publication number is a five-character identifier made up of numbers or upper-case letters. For the Knowledge Warehouse project there were only seven publications to begin with and it was natural to number these 00001 to 00007.

4.1.4. Volume Number

The volume number is a two digit identifier. Initially all Knowledge Warehouse publication modules had only one volume and so "01" was used. Later, some publication modules were split into multiple smaller modules and for these "01" remained the top-level publication modules with the referenced modules being numbered from "02" onwards.

4.1.5. Putting It All Together

When used in file names or URIs the publication module code is preceded by "PMC" and the elements joined together with dashes, for example:

PMC-AIRLINEDOCS-CAG01-00001-01.xml

4.2. Publication Module Entry Title

The <pmEntryTitle> element contains the title for a group of entries in the publication table of contents. On other words, these can be thought of as chapter or section titles.

5. Business Rules

The S1000D specification talks a great deal about business rules which it defines as "decisions that are made by a project or an organization on how to implement S1000D"[[6] [18]]. Initially there was a feeling in the project team that business rules were probably unnecessary for the Knowledge Warehouse project. Of course they would be essential if for developing technical documentation for an aircraft but for general-purpose document management they seemed an unnecessary concern. This thinking was quickly proved wrong once users were let loose on the system for the first time. Business rules are in fact absolutely key to the success of any S1000D project and it is for this reason that the specification talks about them so much, and especially, about the Business Rule Decision Points (BRDP) and Business Rules Exchange (BREX). Without any business rules, authors are free to create whatever content they want, provided that the document remains valid according to the relevant schema.

Table 9. A word of advice..

> Don't be shy of business rules. Business rules are your friends and will makes the lives of everyone involved in the project easier with a little up-front investment in time and effort. Without business rules, you may suddenly find your beautifully crafted S1000D system no longer functions as expected just because a user has entered an incorrect code or has failed to enter required information in a particular field.

5.1. Business Rule Decision Points

The S1000D specification includes a set of Business Rule Decision Points (BRDP). In S1000D issue 4.1 there are over 550 individual BRDP. These decision points are there to help implementers to make important decisions about their project before starting the implementation. Not all decision points are relevant to all projects, so the first step should be to review all the points and select the ones that apply. Some of the decisions to be made will be simple, such as which language to use, and some will require consideration at a broader business level because the decisions made for one project may have an impact on other projects at a later date, for example the model identification code to be used.

For the Knowledge Warehouse project, a proper review of the BRDP was not done prior to the implementation for the reasons mentioned earlier, although many of the decisions still had to be made. With the benefit of hindsight, a proper BRDP review would've been beneficial and prevented some inconsistencies that had to be rectified later. For example there was confusion about whether the country code of modules should be UK or GB and both codes were used in documents. In fact only GB is an ISO valid identifier and formalising this decision in advance would've prevented the wrong code being used and having to be corrected later.

It should be acknowledged however that for those unfamiliar with S1000D, the purpose of a particular BRDP may not become clear until the project is well under way. Therefore only a baseline review of BRDP may be possible initially with further reviews later to understand if rules have become relevant. For the Knowledge Warehouse project, the following BRDP were very important and could probably be considered baseline requirements for any S1000D implementation. Appendix A [17] contains further decision points that were also important for Knowledge Warehouse as well as some that were not considered but which, with the benefit of hindsight, probably should have been.

Table 10. Baseline BRDP for the Knowledge Warehouse project

Identifier	Title
BRDP-S1-00002	List of permitted CAGE codes and/or names of the originator companies to be used for the technical publications
BRDP-S1-00003	Issue of S1000D to be used
BRDP-S1-00005	Publications to be produced
BRDP-S1-00006	Schemas to be used
BRDP-S1-00047	Country and language codes
BRDP-S1-00050	Source of the technical names
BRDP-S1-00070	Use of the element <enterpriseName> and/or the attribute enterpriseCode for the responsible partner company
BRDP-S1-00071	Use of the element <enterpriseName> and/or the attribute enterpriseCode for the responsible originator

Identifier	Title
BRDP-S1-00106	Population of the element <refs>
BRDP-S1-00332	Allocation of Product model identification code
BRDP-S1-00334	Allocation of system difference code
BRDP-S1-00336	Product SNS structure
BRDP-S1-00338	Number of characters in assembly code
BRDP-S1-00344	Use of CAGE code and/or model identification code based ICN
BRDP-S1-00366	Use of a project specific BREX data module
BRDP-S1-00368	Applicable sets of business rules

5.2. Business Rules Exchange (BREX)

Every S1000D data module must refer to a BREX data module. The reference BREX data module defines the business rules that apply to that data module. Because it is a data module it can be rendered and viewed just like any other data module. This means that it not only defines the rules in a computer-readable way but also in a human-readable way as well. It is an excellent example of how S1000D enables you to reuse data modules and have a single source for information.

BREX data modules themselves are layered and each S1000D issue defines its own BREX at the highest level. This means that even if a project-specific BREX has not been defined, data modules will need to reference the S1000D BREX. If a project-specific BREX is defined, and hopefully it is now clear that is strongly recommended, then this BREX will need to refer to the S1000D BREX and will inherit the rules therein. It is possible to layer BREX, perhaps by having a BREX for the organisation and then a BREX for the project which references the organisational BREX and ultimately the S1000D BREX.

The business rules defned in a BREX data module can be divided into three categories: the context rules, the context independent rules and the SNS rules.

5.2.1. Context Rules

Context rules are contained in the <contextRules> element and define permitted values using one or more <objectValue> elements; a permitted structure, using an <objectPath> element that is specified with an XPath expression; and an <objectUse> element which is used to describe and document the rule. These elements are bound together in a <structureObjectRule> element to form a complete rule.

5.2.2. Context Independent Rules

Non-context rules are contained in a <nonContextRules> element and apply throughout the project. They can be used to document rules such as how data modules and publication modules should be reviewed and approved.

5.2.3. SNS Rules

SNS rules define how the Standard Numbering System is to be used in the context of a project.

6. Lessons Learned

Perhaps the single most important lesson learned from the Knowledge Warehouse project is not a revelation but more of a reminder and it could be summarised by the five Ps – Proper Preparation Prevents Poor Performance. The S1000D specification may be daunting but the authors have tried to help implementers with the notion of the Business Rule Decision Points. Reviewing these rules and making appropriate decisions in preparation for an implementation will go a long way to making the project run more smoothly. Enforcing these decision using BREX will ensure that this discipline is maintained, even after the implementation is complete.

6.1. Spreadsheet It!

The other key way to prepare for a project is to engage with the customer or users and help them to design what the project delivery will look like. Following the migration of the Knowledge Warehouse data several other data sets for the same airline were also migrated to S1000D. These migrations were many times easier to perform thanks to the lessons learned from the original project. Perhaps the single most important way that these migrations were different is that the decision making was completely handed over to the customer. Rather than the implementation team taking the source data and trying to reproduce it in S1000D, the customer decided exactly what the S1000D would like when the migration was complete. To facilitate this a list of documents to be migrated was made and placed in a spreadsheet. The customer

then decided what publication, chapter and section the document would appear in when migrated. They also decided what the technical name and information name of each document would be, what the SNS number would be, what model identification and system difference codes would be used. So that ultimately they knew exactly what to expect when the content was migrated. This also allowed the implementation team to highlight any potential problem areas before the migration started. Using this approach was extremely successful, to the extent that the customer knew exactly what to expect. In fact when viewing the content post-migration it appeared so exactly to match their expectation that they had to be persuaded to conduct thorough acceptance testing! "What's the point?" they said "it looks perfect!". In fact, of course, it wasn't perfect and a few changed did need to be made but nothing compared to the restructuring that had to be done for Knowledge Warehouse post-migration.

6.2. Hearts and Minds

You might think that when you have migrated your customer's content to beautiful S1000D, and it all looks great, that your job is done! Think again. You have created an S1000D XML-based system for people who are almost certainly not XML experts, never mind S1000D experts. While they may be experts in the inner workings of an aircraft undercarriage you should not expect them to feel right at home with S1000D. Probably they had achieved a level of comfort with Microsoft Word or Adobe Acrobat but now you are asking them to start again with XML authoring. Let's be clear, there are big advantages to having structured S1000D over unstructured documents but the advantages come at a price. You will need to have rigorous procedures around managing, authoring and publishing content. S1000D is not lightweight because the aviation industry and the military need robust information management tools to be able to ensure flight and equipment safety. You may need to rethink your own business process, perhaps by upskilling a core team of technical authors to manage this content.

You will also need to provide quality training materials for your users, which they can refer to whenever they need. One of the big advantages of S1000D in this regard is that it is a document management system so you can use it can document itself! This means that not only can users access training and materials easily but they can see, through real examples in the system, how to do the things they need to do.

7. Conclusion

The Knowledge Warehouse project implementation was a steep learning curve for all involved. S1000D is not just about XML and being an XML expert does not automatically make you an S1000D expert. S1000D is built on decades of experience in maintaining documentation for complex civil aviation and military systems and because of this, it is a robust and complex system in itself. It has also been designed to meet a very broad spectrum of requirements and it is for this reason that the specification is a bulky and intimidating document.

Perhaps the key to getting the most out of S1000D is in chapter 1.4.1 of the specification[[7] [18]] where it talks about tailoring for a specific project or organisation. I think this is a very helpful way to think about it, especially if you take the analogy of a tailor a little further. We all know that tailors cut cloth and sew the pieces together to create a bespoke item of clothing for the customer. Before getting out the scissors though, the tailor asks what the customer is looking for and then proceeds to take a series of measurements to ensure the finished article is exactly what is requested and that it fits perfectly. The application of S1000D business rules are the way to understand your customer's requirements and get the measurements of your project. With these measurements you can then tailor the specification to fit, leaving out all the parts that aren't relevant. Furthermore, business rules give you the thread to sew the different components of your project together and hold them together securely in the long term.

A. Appendix A

Table A.1. Other important BRDP for the Knowledge Warehouse project

Identifier	Title
BRDP-S1-00008	Possible deliverables
BRDP-S1-00012	Define security classification values and terms (attribute securityClassification)
BRDP-S1-00020	Specify the language
BRDP-S1-00026	Highlighting text
BRDP-S1-00036	Presentation of the issue number and the inwork number on the title page
BRDP-S1-00049	Definition of the issue date
BRDP-S1-00051	Rules for the information names

Identifier	Title
BRDP-S1-00052	Allocation of the information codes and the information names
BRDP-S1-00129	Suitability of multimedia use
BRDP-S1-00130	Permitted types of multimedia
BRDP-S1-00171	Use of the symbols
BRDP-S1-00179	Granularity of data in descriptive data modules
BRDP-S1-00180	Level of depth of descriptive data modules
BRDP-S1-00333	Allow the use of one or several model identification codes
BRDP-S1-00342	Use of the disassembly code variant
BRDP-S1-00343	Use of numeric values in the information code variant
BRDP-S1-00347	Structure and rules for ICN for model identification code based ICN
BRDP-S1-00348	Allocation of responsible partner company codes for model identification code based ICN
BRDP-S1-00349	Security classifications to be used for model identification based ICN
BRDP-S1-00365	Use of the attribute pmIssuer
BRDP-S1-00367	Use of layered BREX data modules
BRDP-S1-00387	Use of applicability
BRDP-S1-00475	Page size
BRDP-S1-00478	Presentation of "Produced by" - "Printed in"
BRDP-S1-00479	Presentation of publication module code
BRDP-S1-00480	Presentation of data module code
BRDP-S1-00481	Presentation of issue date
BRDP-S1-00482	Presentation of page number
BRDP-S1-00496	Presentation of the document title
BRDP-S1-00519	Presentation of change marking of individual table rows
BRDP-S1-00520	Presentation of data module titles in the reference table

Table A.2. BRDP that may have been beneficial to consider for Knowledge Warehouse but were not

Identifier	Title
BRDP-S1-00021	Use of ASD Simplified Technical English
BRDP-S1-00022	Standard dictionary
BRDP-S1-00023	Use of a terminology database or glossary
BRDP-S1-00024	Use of a standard list of abbreviations
BRDP-S1-00025	Units of measurement

References

[1] S1000D Official Homepagehttp://public.s1000d.org/Pages/Home.aspx

[2] "International specification for technical publications using a common source database", Issue No. 4.1, published by ASD/AIA, 2012: Chapter 1.1

[3] ATA e-Business Program: Standards http://ataebiz.org/Pages/standards.aspx

[4] "International specification for technical publications using a common source database", Issue No. 4.1, published by ASD/AIA, 2012: Chapter 4.2

[5] Commercial and Government Entity Program https://cage.dla.mil/

[6] "International specification for technical publications using a common source database", Issue No. 4.1, published by ASD/AIA, 2012: Chapter 2.5

[7] "International specification for technical publications using a common source database", Issue No. 4.1, published by ASD/AIA, 2012: Chapter 1.4.1

Two (and a half) models for markup of bibliographic references

Mark Dunn, Oxford University Press

Abstract

This paper describes two models for semantic markup of bibliographic references, their advantages and disadvantages, and the challenges of automating a conversion from one format to the other. In one format a reference is captured as plain text. The semantic markup (author name, title, publication date, etc) is captured in attributes on the element containing the reference text. The other format is BITS, where each component of a reference is captured in its own element. The BITS model comes in two flavours, <element-citation> and <mixed-citation>, hence the extra half a model in the title. The conversion to BITS needs to support OpenURL linking and the Initiative for Open Citations (I4OC).

1. Background

In late 2015 OUP announced a partnership with Silverchair Information Systems to host its journals and book-based online products.

The new platform accepts books in a flavour of the industry-standard BITS (Book Interchange Tag Suite) data model[1].

OUP book content has been captured for many years in a bespoke data model. Before being loaded to the new platform, this content must be converted to BITS. An XSLT script is being written to automate the conversion.

The OUP data model is broadly compatible with BITS, but one major point of difference is the approach taken to modelling bibliographic references. This paper discusses the approaches to modelling references, and the challenges of converting from the OUP format to BITS whilst maintaining the online functionality that the markup supports.

2. Bibliographic reference formats

A bibliographic reference conveys information that enables a reader to locate another resource. E.g. for a book, the necessary information includes author name, title, and publication date. For a journal article a reference would also include the journal title, volume number, and issue number.

There are a number of standard formats for bibliographic references, including Chicago[2], Harvard[6], and Vancouver[3]. There are various ways in which they differ from one another:

· Spacing and punctuation of components.

· Order of components

· Number of authors listed (before hitting "et al.").

· Style of author name (e.g. "surname, given names", "surname, initials", "initials surname", etc.).

· Style of abbreviating a journal title.

· How the references are indicated in the text (e.g. superscript number, author/date).

The standards run to hundreds of pages of rules. In addition, universities often provide their own interpretations or clarifications of the rules.

Tools such as Zotero[10] make it easy for authors to generate bibliographic references in the format of their choice. Publishers then have to handle the output and turn it into XML.

Example 1. Book, Chicago style

(First author name is in the format "surname, foremanes"; second and subsequent author names are in format "forenames surname".)

Lalloo, Fiona, Bronwyn Kerr, JM Friedman, and Gareth Evans. *Risk assessment and management in cancer genetics* (Oxford: Oxford University Press, 2005)

Example 2. Book, Harvard style

(All author names are in format "surname, initials"; publication date precedes book title.)

Lalloo, F., Kerr, B., Friedman, J., and Evans, G. (2005) *Risk assessment and management in cancer genetics* Oxford, Oxford University Press

Example 3. Book, Vancouver style

(Author names are in format "surname initials"; no italicization of title.)

Lalloo F, Kerr B, Friedman J, Evans, G. Risk assessment and management in cancer genetics. Oxford: Oxford University Press; 2005

Example 4. Journal article, Chicago style

(Double quote marks around article title.)

Salomone, A, A Bozzo, D Di Corcia, E Gerace, and M Vincenti. "Occupational Exposure to Alcohol-Based Hand Sanitizers: The Diagnostic Role of Alcohol Biomarkers in Hair." *Journal of Analytical Toxicology* 42, no. 3 (1 April 2018): 157–162, https://doi.org/10.1093/jat/bkx094

Example 5. Journal article, Harvard style

(Single quote marks around article title.)

Salomone, A., Bozzo, A., Di Corcia, D., Gerace, E., and Vincenti M. (2018) 'Occupational exposure to alcohol-based hand sanitizers: the diagnostic role of alcohol biomarkers in hair', *Journal of Analytical Toxicology* 42(3), 157–162, available: https://doi.org/10.1093/jat/bkx094

Example 6. Journal article, Vancouver style

(Abbreviated journal title.)

Salomone A, Bozzo A, Di Corcia D, Gerace E, Vincenti M. Occupational exposure to alcohol-based hand sanitizers: the diagnostic role of alcohol biomarkers in hair. J Anal Toxicol. 2018 Apr 1;42(3):157–162, doi:10.1093/jat/bkx094

Example 7. Journal article, style rendered by OUP Academic platform

(Generated from JATS article metadata.)

A Salomone, A Bozzo, D Di Corcia, E Gerace, M Vincenti; Occupational Exposure to Alcohol-Based Hand Sanitizers: The Diagnostic Role of Alcohol Biomarkers in Hair, *Journal of Analytical Toxicology*, Volume 42, Issue 3, 1 April 2018, Pages 157–162, https://doi.org/10.1093/jat/bkx094

3. Online functionality of bibliographic references

The benefit in online content of a bibliographic reference is having it link directly to the resource. Journal articles typically have DOIs[5], and a reference to a journal article might include the DOI. This can easily be parsed into a link which will be resolved to the URL at which the referenced article can be found. Many journal articles (and references to them) also include PubMed[9] IDs, which work in a similar way.

Sometimes the best that can be done with a bibliographic reference is to pass the components to a search tool, as parameters to a URL.

The components of a reference must be identified in the XML in order to generate accurate search URLs.

Example 8. OpenURL search

http://sfx-demo.exlibrisgroup.com:3210/demo?sid=oup:oxmed&genre=book&title=Risk%20assessment%20and%20management%20in%20cancer%20genetics&aulast=Lalloo&aufirst=F&date=2005

Example 9. COPAC search

http://copac.ac.uk/search?ti=Risk+assessment+and+management+in+cancer+genetics&au=Lalloo

Example 10. Google Preview

https://www.google.com/search?q=Risk+assessment+and+management+in+cancer+genetics&btnG=Search+Books&tbm=bks&tbo=1

Example 11. WorldCat search

http://worldcat.org/search?q=ti%3ARisk+assessment+and+management+in+cancer+genetics&qt=advanced&dblist=638

4. OUP data model

When OUP was first developing its data model, the digital content was created by converting scholarly monographs that had already been published in print. The intention was to sell the online content as a way of making these books more widely available.

The online version had to be as faithful as possible to the original print version. This meant that bibliographic references had to be captured as plain text, as submitted by the author. Each reference was wrapped in a `<bibItem>` element.

Having the references link to library catalogues (via OpenURL[8]) was desirable, so we chose to identify the components of a reference by copying them into attribute values on the `<bibItem>` element.

OpenURLs and other search links are generated on the platform by parsing these attributes.

This model also lets us add value to abbreviated references (e.g. "Ibid.")

Example 12. OUP data model for a book reference

(Using Vancouver style for the text of the reference.)

```
<bibItem
  class="book"
  author="Lalloo F|Kerr B|Friedman J|Evans G"
  title="Risk assessment and management in cancer genetics"
  date="2005"
  place="Oxford"
  publisher="Oxford University Press"
  >
    Lalloo F, Kerr B, Friedman J, Evans G.
    Risk assessment and management in cancer genetics.
    Oxford: Oxford University Press; 2005
</bibItem>
```

Example 13. OUP data model for a journal article reference

(Using Vancouver style for the text of the reference.)

```
<bibItem
  class="journalArticle"
  author="Salomone A|Bozzo A|Di Corcia D|Gerace E|Vincenti M"
  title="Occupational exposure to alcohol-based hand sanitizers:
      the diagnostic role of alcohol biomarkers in hair"
  date="2018"
  vol="42"
  journalIssue="3"
  page="157"
  pageLast="162"
  doiTarget="10.1093/jat/bkx094"
  >
    Salomone A, Bozzo A, Di Corcia D, Gerace E, Vincenti M.
    Occupational exposure to alcohol-based hand sanitizers:
      the diagnostic role of alcohol biomarkers in hair.
    J Anal Toxicol. 2018 Apr 1;42(3):157–162,
    doi:10.1093/jat/bkx094
</bibItem>
```

5. BITS data model

The BITS data model takes the approach of wrapping each component of a reference in an element describing its meaning.

There are two different flavours of this model.

- The `<element-citation>` model contains only elements. Punctuation of the components to style a reference in any of the standard formats is a task left to rendering engines.

- The `<mixed-citation>` model consists of mixed content. Punctuation must be included among the content of a reference.

Example 14. BITS data model for a book reference

With <element-citation>:

```
<element-citation>
  <name>
    <surname>Lalloo</surname>
    <given-names>F</given-names>
  </name>
  <name>
    <surname>Kerr</surname>
    <given-names>B</given-names>
  </name>
  <name>
    <surname>Friedman</surname>
    <given-names>J</given-names>
  </name>
  <name>
    <surname>Evans</surname>
    <given-names>G</given-names>
  </name>
  <year>2005</year>
  <source>Risk assessment and management in cancer genetics</source>
  <publisher-loc>Oxford</publisher-loc>
  <publisher-name>Oxford University Press</publisher-name>
</element-citation>
```

With <mixed-citation> (Vancouver style):

```
<mixed-citation>
  <string-name><surname>Lalloo</surname> <given-names>F</given-names></string-name>,
  <string-name><surname>Kerr</surname> <given-names>B</given-names></string-name>,
  <string-name><surname>Friedman</surname> <given-names>J</given-names></string-name>,
  <string-name><surname>Evans</surname> <given-names>G</given-names></string-name>.
  <source>Risk assessment and management in cancer genetics</source>.
  <publisher-loc>Oxford</publisher-loc>:
  <publisher-name>Oxford University Press</publisher-name>;
  <year>2005</year>
</mixed-citation>
```

Example 15. BITS data model for a journal article reference

With <element-citation>:

```
<element-citation>
  <name>
    <surname>Salomone</surname>
    <given-names>A</given-names>
  </name>
  <name>
    <surname>Bozzo</surname>
    <given-names>A</given-names>
  </name>
  <name>
    <surname>Di Corcia</surname>
    <given-names>D</given-names>
  </name>
```

```
    <name>
      <surname>Gerace</surname>
      <given-names>E</given-names>
    </name>
    <name>
      <surname>Vincenti</surname>
      <given-names>M</given-names>
    </name>
    <article-title>Occupational exposure to alcohol-based hand sanitizers:
      the diagnostic role of alcohol biomarkers in hair</article-title>
    <source>J Anal Toxicol</source>
    <year>2018</year>
    <volume>42</volume>
    <issue>3</issue>
    <fpage>157</fpage>
    <lpage>162</lpage>
    <pub-id pub-id-type="doi">10.1093/jat/bkx094</pub-id>
</element-citation>
```

With <mixed-citation> (Vancouver style):

```
<mixed-citation>
    <string-name><surname>Salomone</surname> <given-names>A</given-names></string-name>,
    <string-name><surname>Bozzo</surname> <given-names>A</given-names></string-name>,
    <string-name><surname>Di Corcia</surname> <given-names>D</given-names></string-name>,
    <string-name><surname>Gerace</surname> <given-names>E</given-names></string-name>,
    <string-name><surname>Vincenti</surname> <given-names>M</given-names></string-name>.
    <article-title>Occupational exposure to alcohol-based hand sanitizers:
      the diagnostic role of alcohol biomarkers in hair</article-title>.
    <source>J Anal Toxicol</source>.
    <year>2018</year> <month>Apr</month> <day>1</day>;
    <volume>42</volume>(<issue>3</issue>):<fpage>157</fpage>–<lpage>162</lpage>,
    doi:<pub-id pub-id-type="doi">10.1093/jat/bkx094</pub-id>
</mixed-citation>
```

6. Converting the OUP data model to BITS

The first proposal was to throw away the content of the OUP `<bibItem>` and turn the attribute values into elements within a BITS `<element-citation>` element.

The `<element-citation>` model is an ideal destination; clean, and easy to capture. But there are difficulties in implementing this for OUP content.

The platform needs a robust rendering engine to apply spacing and punctuation to an `<element-citation>` element. OUP content has many different types of reference, including:

- book
- book chapter
- journal article
- newspaper/magazine
- conference
- archive
- web link
- catalogue
- patent
- thesis

- personal correspondence with the author

- etc.

(Not to mention references to many kinds of legal documents, treaties, legislation, etc.)

Developing and testing a rendering engine for all these styles is not a trivial task.

There's also a proportion of references in which the attribute values have been captured incorrectly. This is currently invisible to users of the platform (until they click on a link generated from the attribute values), but would be exposed if the attribute values were converted to element content.

There are references where the author does not follow a standard, e.g. inserting some discursive text. The BITS model includes a <comment> element for holding this, but in OUP's data model there is no attribute to capture this text.

Example 16. Non-standard references

Klaassen, C. D., Ed. (2001). *Casarett and Doull's Toxicology: The basic science of poisons.* McGraw-Hill Medical, New York. This broad detailed introduction to toxicology is a good reference for the clear presentation of the concepts, methods, and common toxic materials and contains many nice examples.

DANTE ALIGHIERI, *The Divine Comedy*; we recommend the translation by D. L. Sayers (3 vols.; Harmondsworth: Penguin Books, 1950–62).

The best general treatments of the US Constitution include the classic work by Alfred Kelly, Winfred Harbison, and Herman Belz, *The American Constitution: Its Origins and Developments*, 7th ed. (New York: W. W. Norton, 1991) and Melvin I. Urofsky and Paul Finkelman, *The March of Liberty: A Constitutional History of the United States*, 3rd ed. (New York: Oxford University Press, 2011). Good short treatments are Michael Les Benedict, *The Blessings of Liberty: A Concise History of the Constitution of the United States* (Lexington, MA: D. C. Heath, 1996) and David J. Bodenhamer, *The Revolutionary Constitution* (New York: Oxford University Press, 2012).

OUP decided to keep the requirement that references should be presented as the author intended, which rules out the `<element-citation>` model and leaves us with `<mixed-citation>`.

7. Outline of the XSLT

The approach we adopted was to pass the content of the <bibItem> element through a template a number of times, each time looking for a different attribute to match in the text.

Example 17. Stages of conversion

Original content:

```
Deborah Furet How Illusions Pass, 176-196, 234-247. London: Gerald Duckworth, 1987
```

After first pass:

```
Deborah Furet <title>How Illusions Pass</title>, 176-196, 234-247.
London: Gerald Duckworth, 1987
```

After second pass:

```
Deborah <surname>Furet</surname> <title>How Illusions Pass</title>,
176-196, 234-247. London: Gerald Duckworth, 1987
```

After third pass:

```
Deborah <surname>Furet</surname> <title>How Illusions Pass</title>,
176-196, 234-247. London: Gerald Duckworth, <year>1987</year>
```

The template for the original <bibItem> element is:

```
<xsl:template match="bibItem[
```

```
          @class = 'bookChapter'
          or @class = 'book'
          or @class = 'journalArticle'
          ]">
          <xsl:variable name="content" as="node()+" select="child::node()"/>

          <xsl:variable name="content-plus-title" as="node()+">
            <xsl:choose>
              <xsl:when test="@title">
                <xsl:call-template name="getElement">
                  <xsl:with-param name="class" select="@class" tunnel="yes"/>
                  <xsl:with-param name="attributeName" select="'title'" tunnel="yes"/>
                  <xsl:with-param name="stringToMatch" select="@title" tunnel="yes"/>
                  <xsl:with-param name="matched" select="false()" tunnel="yes"/>
                  <xsl:with-param name="nodesToCheck" select="$content"/>
                </xsl:call-template>
              </xsl:when>
              <xsl:otherwise>
                <xsl:copy-of select="$content"/>
              </xsl:otherwise>
            </xsl:choose>
          </xsl:variable>

          <xsl:variable name="content-plus-name" as="node()+">
              <!-- call to "getElement" but parameter $nodesToCheck is $content-plus-title -->
          </xsl:variable>

          <xsl:variable name="content-plus-date" as="node()+">
              <!-- call to "getElement" but parameter $nodesToCheck is $content-plus-name -->
          </xsl:variable>

          <xsl:copy>
            <xsl:apply-templates select="@*"/>
            <xsl:copy-of select="$content-plus-date"/>
          </xsl:copy>

      </xsl:template>
```

The content of the <bibItem> element goes through several stages within the template. Passing from one stage to the next involves a call to a template "getElement" to find a particular attribute value within the content and wrap it in the corresponding BITS element. The output of the first stage is passed as the input to the next, and so on, until the output of the last stage is copied into the output XML tree.

The "getElement" template that does the matching is:

```
<xsl:template name="getElement">
    <xsl:param name="class" as="xs:string" required="yes" tunnel="yes"/>
    <xsl:param name="attributeName" as="xs:string" required="yes" tunnel="yes"/>
    <xsl:param name="stringToMatch" as="xs:string" required="yes" tunnel="yes"/>
    <xsl:param name="matched" as="xs:boolean" tunnel="yes"/>
    <xsl:param name="nodesToCheck" as="node()+"/>

    <xsl:variable name="thisNode" as="node()" select="$nodesToCheck[1]"/>
    <xsl:variable name="remainingNodes" as="node()*"
    select="$nodesToCheck except $thisNode"/>

    <xsl:variable name="nodeAsString" as="xs:string" select="string($thisNode)"/>

    <xsl:variable name="matches" as="xs:boolean"
    select="oupdtg:hasMatch($nodeAsString,$stringToMatch)"/>

    <!--
```

```
        Code to look at $thisNode (the first node)
        Either copy it (if $stringToMatch is not found,
        or if it has been matched already)
        or wrap part of the text node matching $stringToMatch
        in an element appropriate to the $attributeName
    -->

    <xsl:if test="count($remainingNodes) != 0">
      <xsl:call-template name="getElement">
        <xsl:with-param name="matched" select="$matched or $matches" tunnel="yes"/>
        <xsl:with-param name="nodesToCheck" select="$remainingNodes"/>
      </xsl:call-template>
    </xsl:if>

</xsl:template>
```

The parameters to the template are:

Parameter	Description
class	The type of item that is being referenced (e.g. book, journal article).
attributeName	The name of the attribute whose value is being sought in the content (e.g. "author", "title"). The value of this and the $class parameter determine the name of the element that is wrapped round a matching string.
stringToMatch	The value (author name, article title, etc) that the template is looking for in the content.
matched	A boolean variable, whose value is "true()" if the string has already been matched in the content.
nodesToCheck	The remaining content of the bibItem. The template parses one node at a time, and then recursively calls itself with the remaining nodes passed as this parameter.

8. Limitations of the conversion

We could in theory add more passes through "getElement" to capture the publisher name, volume number, issue number, etc. But there are diminishing returns. There's an increasing risk of inaccurate results. For example, one number looks very much like another within a string.

And adding more elements to the citation doesn't necessarily improve the results from an OpenURL link. We've noticed that including more parameters in an OpenURL can result in the link failing to match an item in a library catalogue. Unless each parameter matches a corresponding field in the catalogue metadata, the OpenURL will not return a result. The sweet spot for references to books seems to be surname of first author, title, and publication date.

We have so far chosen only to identify the first contributor to a referenced work, and only their surname. The format of the @author attribute of the <bibItem> element is "surname, forenames". This doesn't always match the format in the text of the reference, which could be "forenames surname". We have to weigh the value of identifying other contributors and their forenames against the effort of enhancing the conversion templates.

Limiting the conversion to just a few fields gives us a good chance of capturing enough information with enough accuracy to support OpenURL linking.

This is not enough to support the Initiative for Open Citations (I4OC)[7] fully, since the markup is incomplete. I4OC is an initiative to promote availability of citation data. This is facilitated by depositing bibliographic references as part of the metadata associated with a DOI. CrossRef[4] supports the deposit of unstructured references, which allows us to provide limited support for I4OC, but ideally we would provide fully structured references.

One benefit of the OUP data model that we are losing in the conversion to BITS is the ability to handle partial references. An author referring repeatedly to the same work may use the word "Ibid." to save space on subsequent references, or substitute a couple of em-dashes for the author name. Using a <bibItem> for these allows us to capture the full information even on a partial reference, but the BITS model doesn't let us do that. This situation is increasingly rare, as authors are discouraged from using these styles.

Bibliography

[1] *Book Interchange Tag Set (BITS)*: https://jats.nlm.nih.gov/extensions/bits/

[2] *Chicago Manual of Style*: http://www.chicagomanualofstyle.org/home.html

[3] *Citing Medicine (2nd Edition)*: https://www.ncbi.nlm.nih.gov/books/NBK7256/

[4] *CrossRef*: https://www.crossref.org/

[5] *DOI*: https://www.doi.org/

[6] *Harvard Style (Imperial College London guide)*: https://www.imperial.ac.uk/media/imperial-college/administration-and-support-services/library/public/harvard.pdf

[7] *Initiative for Open Citations (I4OC)*: https://i4oc.org/

[8] *OpenURL (ANSI/NISO Z39.88-2004 (R2010))*: https://www.niso.org/publications/z3988-2004-r2010-openurl-framework-context-sensitive-services

[9] *PubMed*: https://www.ncbi.nlm.nih.gov/pubmed/

[10] *Zotero*: https://www.zotero.org/

The Cryptic Crossword Corpus Project: first steps in establishing a markup vocabulary

Bethan S. T. Tovey

Abstract

In a quick crossword, the relationship between clue and answer is usually simply definitional: the clue offers a definition or description of, or a synonym for, the answer. In a cryptic crossword, the relationship between clue and answer is significantly more complex, and may be one of many types. Anagrams, puns, hidden words, deletions, reversals, and other techniques are used, often in combination. Every word in a clue does one of three jobs: a) defining the answer (a synonym or definition of the answer word(s)); b) producing part or all of the answer word; or c) indicating how the words in b) are to be manipulated. It follows that each part of an answer will relate back to one or more words in the clue from category b), while the whole answer will relate to the word(s) in category a).

This paper describes the early work of creating a markup scheme which relates each part of a cryptic crossword clue to the relevant part of its answer, as well as giving full linguistic detail for each word. The aim of this project is to create a corpus of cryptic clues and answers which can be explored to answer linguistic questions.

1. Introduction

Cryptic crossword clues form a textual genre that remains surprisingly under-theorized from almost any relevant linguistic standpoint (Pham, 2016). Two papers by semioticians stand out as contributions to the understanding of the genre (Greimas, 1967; Vântu, 1991), as does a recent corpus-linguistic analysis of crossword puzzles as a distinct register (Pham, 2016). Articles examining the linguistic playfulness and ambiguity of cryptic clues may be added to these as examples of introductory explorations (Cleary, 1996; Coffey, 1998; Rambousek, 2004). Beyond this, many papers published about crosswords (whether quick or cryptic) tend to be psychological investigations of crossword solvers' intelligence or other cognitive abilities, or the role of crosswords (and other puzzles) in cognitive development and in preventing cognitive decline (e.g. Underwood, 1994; Lewis, 2006; Friedlander, 2009; Nickerson, 2011; Moxley, 2015; Friedlander, 2016). The other common type of paper is the description of computational approaches to writing or solving crossword clues (e.g. Williams, 1979; Smith, 1986; Littman, 2000; Hardcastle, 2007).

Could it be that the paucity of linguistic analyses of the cryptic crossword can be attributed to a lack of interesting linguistic features? After all, an entire puzzle's worth of crossword clues contains relatively few words in total, and there is little discursive linkage between individual clues. In the terms defined by the foundational text-linguistic work by Halliday (1976), a set of cryptic crossword clues lacks *texture*: the network of linguistic relationships between sentences that makes a coherent *text*. A set of clues is a set of fragments, but not a text.

Nevertheless, a number of authors have argued for the merit of the cryptic crossword as an object of linguistic study. From the perspective of semiotics, Greimas (1967) suggests that the crossword should be analysed on the same terms as other types of aesthetic "communication différée" (p. 799) (*delayed communication*). By this, he means uses of language (like poetry) which are non-direct, and which necessitate a process of decoding in order to achieve communication between the creator (known as the "setter" of a cryptic crossword) and the audience. He notes that the absence of grammatical predicates in French crossword clues is the type of feature upon which a typology of the genre might be built. Since the predicate carries many of the temporal specifics of a sentence, as well as indicating the role of elements such as subject and object, it is no surprise to find that the lack of a predicate is also a feature of the English cryptic clue. Stratmann (1978) calls attention to the relationship between the cryptic clue's encoding of meaning and the literary tradition of Lewis Carroll and James Joyce. He links the success and complexity of the cryptic crossword clue to specific linguistic features of English, calling attention to the abundance of short words and the lack of mandatory inflectional endings, aspects of the language that make flexible clue-writing much easier. Indeed, Connor (2013b) argues that English is "the best crosswording language" (p.114) as a result of its large vocabulary and rich range of synonyms (the result of large-scale borrowing from other languages over more than a millennium). The cryptic crossword clue benefits from the range of potential ambiguity residing in English synonyms, as well as in the comparative lack of grammatical specificity that Stratmann (1978) identifies. In an uninflected language like English, it is no doubt easier to have a word that looks like (say) a noun, but is intended by the setter as a verb; there are few required word-endings that would give the game away.

Genre-theoretical approaches to the crossword have also been successfully attempted. Vântu (1991), in an exploration of Romanian crossword clues, argues that they may best be analyzed as wordplay or as jokes. Cleary (1996) calls English cryptic crosswords "exercises in constructed ambiguity" (p.15). Like Vantu, he sees in them a source for study of wordplay and linguistic humour, as well as of the construction and comprehension of meaning. Finally, Pham's, (2016) work on register and intertextuality is perhaps the fullest exploration of the crossword in terms of linguistic theory. Using a corpus of clues and answers, she argues that the use of intertextual reference is a distinctive feature of the crossword as a genre. She also establishes the distinctiveness of cryptic and non-cryptic crosswords as sub-genres.

Evidently, there are sufficient grounds for considering the cryptic crossword as an interesting linguistic object. The Cryptic Crossword Corpus Project (CCCP) aims to provide a corpus-linguistic basis for further work on this genre. The project will gather clues from a range of different publications and crossword compilers, including the big names such as *The Times* and *The Guardian*, but also smaller and (perhaps) more esoteric publications such as the satirical current-affairs magazine *Private Eye* and the socialist newspaper *Morning Star*. It may also prove possible to include clues from cryptic crosswords outside Britain; for example, *The Sydney Morning Herald* and *The Hindu* both set a cryptic crossword, as (occasionally) does *The New York Times*. For the moment, however, the CCCP will focus on British crosswords.

2. A brief explanation of cryptic crosswords

Although crosswords are now to be found in many of the world's languages, for the most part these are "quick" crosswords, whose clues generally consist of a simple synonym of the target answer (Connor, 2013b). The first quick crossword is generally agreed to have been published in New York in 1913; the British cryptic was an offshoot, first found in the 1930s (Arnot, 1981). The cryptic is largely a British phenomenon; Connor (2013a) suggests that this is partly attributable to a British taste for "imagery of the nudge-and-wink variety", toilet humour, and mild profanity, but also to a lack of other solvers from whom to learn the conventions of the cryptic clue. Learning how to solve cryptic crossword clues is a matter of practice, and of learning the rules by which words are encoded into clues (Stratmann, 1978). In order

to understand the specific markup needs of the CCCP, it will therefore be necessary to take a brief tour of how target answers may be encoded by cryptic crossword setters.

A quick clue for the location of this conference might simply read "Capital city (6)" (the number in parentheses indicates how many letters are in the target answer). There are, of course, a number of capital cities with six letters; the quick-crossword solver relies on the intersecting words to provide some of the letters needed to narrow down the possibilities until the correct target is found. The cryptic-crossword solver, in contrast, should be left in no doubt of the correct answer, even if there are no intersecting letters available to confirm it. This is achieved by careful writing of clues that encode each target answer in two ways.

The commonest structure for a cryptic clue consists of the *definition* and the *subsidiary indication*. The definition provides some kind of synonym or narrative definition of the target; the subsidiary indication encodes the target in some other way. A frequent type of subsidiary indication is the *charade*, in which letters or groups of letters from the target are clued in sequence in the subsidiary indication. If we take our "capital city" example, a charade clue might read as follows: "Left Ontario with DeLillo for UK city" (6). The clue breaks down as follows: *L*, a common abbreviation for "left"; *ON*, the postal abbreviation for "Ontario"; *DON*, the first name of the American novelist DeLillo. Put together, these fragments provide the target *LONDON*. "UK city", of course, is the definition. Notice the words "with" and "for". These are not mere filler. Each one has meaning in constructing the clue: "with" indicates that the fragments can be placed together, and "for" is a common indicator that what follows is the definition of the target. I will call these types of words "metalanguage".

The function of the metalanguage is more prominent in this clue: "City bird around nine doves, initially (6)". Here, the clue's first word is the definition, and all that follows is the subsidiary indication. We are looking for a word meaning "bird" which can be placed around the initial letters of "nine doves". The bird is *LOON*, and if we place *ND* between its letters we again arrive at the target *LONDON*. Again, every single word in the clue serves a purpose. Finally, let us consider the following clue: "Melon donut in possession of Julie (6)". Here we need to know that Julie London was a famous singer, in order to understand that "Julie" provides the definition component of the clue. The subsidiary definition is what is known as a "container" type: the letters of the target are inside ("in possession of") the words *meLON DONut*.

These latter two clues show the ambiguity inherent in most cryptic clues, and how linguistic features are exploited to create that ambiguity. The apparent noun phrase "city bird" leads the reader away from the necessary understanding that the split between definition and subsidiary indication happens between the two words. We can see, also, that the cryptic crossword setter is permitted to play fast and loose with punctuation: a more honest punctuation of the clue as "City: bird around nine doves initially" would make the roles of the various words clearer. Similarly, the metalanguage in the final clue is most easily read as suggesting that the donut is in Julie's possession, whereas the solution requires understanding that "in possession of" works in reverse here. Either reading is perfectly syntactically allowable in English, but the more obvious reading is the wrong one for finding the target answer.

These examples, I hope, portray a distinctive characteristic of a well-written cryptic clue: the solver should be left in no doubt that her answer is correct (Pham, 2016). The subsidiary indication makes it clear that no other answer is possible, by providing specific (albeit heavily encoded) instructions about the physical makeup of the target answer (Coffey, 1998). Not all clues consist of a definition and subsidiary indication, however. So-called "double-definition" clues are also popular. These offer two definitions, incorporated into an apparently single syntactic unit, which approach the target answer from two different directions. An example from *The Times* will exemplify this type of clue: "Alert goalkeeper may dive thus (2, 3, 4)". The answer is "ON THE BALL", for which "alert" is a synonym, and "goalkeeper may dive thus" is a descriptive definition. Once the solver has hit upon this answer, it is quite clear that nothing else will fit the bill; there is no other phrase that is defined by both of the definitions. Another example, this time from the *Daily Mail*, shows the dangers of attempting this type of clue: "Guess there's no proof for it (10)". The answer, "CONJECTURE", is not particularly obvious. The two definitions, the synonym "Guess" and the descriptive "there's no proof for it" could apply to many similar words. It will be interesting to see whether intuitions about what makes for a good and a bad clue can be supported by linguistic evidence. For example, it seems possible that a good double-definition clue needs to have a certain amount of semantic distance between the two definitions. If these are too close, there's a risk (as in the Daily Mail example) that the clue ends up being too broad. It is the intersection of apparently quite different synonyms or definitions which both apply to a single target that makes for a good clue.

3. Developing the markup vocabulary for CCCP

The previous section of this paper has given an outline of a sample of the elements that are used to construct a cryptic crossword clue. In developing a markup vocabulary for CCCP, it has been necessary to consider how these elements (as well as others not discussed above for reasons of space) relate to each other, and which of them are required in order to give a comprehensive account of the structure of a clue. There are, of course, elements above the level of the clue that could also be captured, such as the source of the crossword, the name of the setter, the date on which the crossword appeared, and so on.

Making decisions about how much detail to include required some predictions about the questions that might be asked of the corpus data at a later stage of the project. Cleary (1996) suggested some possible avenues for research, including examining differences between setters, and the use of polysemous words both in clues and as targets. To this might be added comparison of the "house styles" of various newspapers, comparison of the language of clues at different points in time, and the use of words with ambiguous part of speech. Evidently, these questions require marking up not only the structural features of the clue itself, but also of the name of the publication, the setter, the date, and the parts of speech of individual words. Since I am also interested in how various fragments of the target answers are commonly (or, indeed, uncommonly) represented in the subsidiary indications, there also needs to be a way to link each part of the clue to the corresponding part of the answer.

The following example shows a single clue and answer pair within the root `<corpus>` element:

```
<corpus>
    <publication pub="independent" type="anthology" ISBN="0550101756" pubdate="2005">
        <puzzle id="1" setter="Aelred">
            <item id="4">
                <clue>
                    <subsidiary>
                        <source class="definition" id="1">
                            <word pos="JJ">Feeble</word>
                            <word pos="NN">type</word>
                        </source>
                        <meta class="locator" type="concatenation">
                            <word pos="VVZ">adopts</word>
                        </meta>
                        <source class="translation" id="2">
                            <word pos="DT">the</word>
                        </source>
                        <meta class="operator" type="translation">
                            <word pos="JJ">French</word>
                        </meta>
                    </subsidiary>
                    <def type="hypernym">
                        <word pos="NN">headdress</word>
                    </def>
                </clue>
                <solution words="1" letters="6" text="wimple" pos="NN">
                    <unit id="1">WIMP</unit>
                    <unit id="2">LE</unit>
                </solution>
            </item>
        </puzzle>
    </publication>
</corpus>
```

(N.B. for ease of reading, part-of-speech tagging is omitted from all examples below).

The first element within the root, `<publication>`, captures the published source for the crosswords. In this case, the source is an anthology, so the ISBN and the publication date are also captured, as is the publication - in this case, *The Independent*. Within this element, the first `<puzzle>` element has an id attribute (to indicate its place within the anthology) and a setter attribute. This attribute must often have the value "anon", because many anthologies fail to identify the specific setter of each puzzle. Each `<puzzle>` element contains a number of `<item>` elements; these contain the clue and answer pairs.

The clue in the above example has the structure defined in section 2 above: a subsidiary indication and a definition. These are captured inside the `<clue>` element, as `<subsidiary>` and `<def>`. The attribute type on `<def>` indicates whether the definition element is a straight synonym, a phrase, a narrative definition, or some other kind of semantic relative of the target answer; in this case, the definition is a hypernym (a term that is more general than the target answer). This information will be useful for exploring questions of how different kinds of definition might affect clue difficulty.

The child elements of `<subsidiary>` are `<source>` (for the words that are manipulated to restructure the target answer) and `<meta>` (for metalanguage). The `source` attribute class indicates how the word must be manipulated in order to produce an answer (or answer fragment). In this example, the first `<source>` is a definition of its corresponding

fragment, and the second must be translated. The attribute class on `<meta>` indicates the function performed by the metalanguage. In this case, the first example is of the "locator" class, and the "concatenation" subtype, which indicates that the result of the second `<source>` is placed after the first. The second example is an operator, and an additional type attribute tells us that the operation required is translation. Finally, each individual word in the clue is enclosed in a `<word>` element, with an attribute indicating part of speech. The part-of-speech tagset used is the Penn Treebank tagset as modified for use in the Sketch Engine corpus software (Sketch Engine). The `<solution>` element contains basic information such as number of words, number of letters, and the plain text of the target answer, as well as its part of speech (if that can be determined). The target itself is then broken down into `<unit>` elements, each of which has an id attribute whose value corresponds to the id of the `<source>` element from which it was produced. For a double-definition clue, the markup looks a little different:

```
<item id="13">
    <clue>
        <def type="synonym">Alert</def>
        <def type="descriptive">goalkeeper may dive thus</def>
    </clue>
    <solution words="3" letters="9" text="on the ball" pos="PHRASE">
        <unit>ON THE BALL</unit>
    </solution>
</item>
```

Here there are two `<def>` elements, and the single `<unit>` element is not given an id attribute, as it is assumed that definitions always point towards the answer.

Clues with "container" type locator class metalanguage generally require embedding one `<unit>` within another in the answer. This is easily achieved by making it legal for one `<unit>` to have another as a child element:

```
<item id="6">
    <clue>
        <def type="descriptive">Irish leader</def>
        <subsidiary>
            <source class="anagram" id="1">is a cheat</source>
            <punct>,</punct>
            <meta class="operator" type="anagram">worried</meta>
            <meta class="locator" type="container">about</meta>
            <source class="abbreviation" id="2">nothing</source>
        </subsidiary>
    </clue>
    <solution words="1" letters="9" text="Taoiseach" pos="NP">
        <unit id="1">TA<unit id="2">O</unit>ISEACH</unit>
    </solution>
</item>
```

Another distinctive markup requirement is found in *The Sun*, which publishes "Two-Speed Crosswords". The same grid of target answers is offered with two sets of clues: one quick, and one cryptic. On the principle that it is always better to capture too much information than too little, I decided to add the quick clues in an initial `<quick>` element for these crosswords:

```
<item id="2">
    <clue>
        <quick>On fire</quick>
        <subsidiary>
            <source class="literal" id="1">A</source>
            <source class="synonym" id="2">fair</source>
        </subsidiary>
        <def type="synonym">land</def>
    </clue>
    <solution words="1" letters="6" text="alight" pos="VV">
        <unit id="1">A</unit>
        <unit id="2">LIGHT</unit>
    </solution>
</item>
```

4. Conclusion

The markup vocabulary I have sketched out above is by no means fixed at this stage. As more clues from more sources are added to the corpus, it may be that changes are needed in order to accommodate the quirks of a particular author. There are also some issues for which I am not sure that satisfactory solutions have yet been found. Perhaps the largest of these issues is that of reference. Some clues may refer to other clues within the same puzzle, using the answer to another clue to construct their own definitions or subsidiary indications. It is clear that there needs to be a way to indicate such cross-reference within a puzzle. But what about exophoric reference (reference to things outside the text)? If we want to understand how linguistic differences affect the difficulty level of various puzzles, for example, we might take a discourse-analytic perspective and consider how the clues reference culture outside the bounds of the puzzle itself. For example, the "LONDON" clues discussed above refer to two cultural figures: Don DeLillo and Julie London. These are evidently figures from different domains - literature and popular music, respectively. Is this information useful? If so, how can it be captured? Depth of information is also a question: is it enough to specify the domain from which these figures come, or would it also be useful to know the period in which they were active? Should their status as real-life figures (as opposed to fictional characters) also be captured?

The answer to these questions, and to others that will undoubtedly arise, will need a certain amount of trial and error, and answers will emerge during the process of marking up more and more clues. Evidently, the initial stages outlined here are just the beginning of what promises to be a complex but rewarding project.

Bibliography

[Arnot 1981] Arnot, M. (1981). *A history of the crossword puzzle*. London: Macmillan.

[Cleary 1996] Cleary, J. (1996). Misleading contexts: the construction of ambiguity in the cryptic crossword clue. *Edinburgh Working Papers in Applied Linguistics*, 7, 14–29.

[Coffey 1998] Coffey, S. (1998). Linguistic aspects of the cryptic crossword. *English Today*, 14(1), 14–18.

[Connor 2013a] Connor, A. (2013a, December). *Are cryptic crosswords too rude for americans?* Retrieved June 4, 2018, from https://www.theguardian.com/crosswords/crossword-blog/2013/ dec/18/cryptic-crosswords-too-rude-for-americans-puzzle

[Connor 2013b] Connor, A. (2013b). *Two girls, one on each knee*. London: Penguin.

[Friedlander 2009] Friedlander, K. J. & Fine, P. (2009). Expertise in cryptic crossword performance: an exploratory survey. In A. Williamon, S. Pretty, & R. Buck (Eds.), *Proceedings of the international symposium on performance science, Auckland* (pp. 279–284). European Association of Conservatoires (AEC). Utrecht.

[Friedlander 2016] Friedlander, K. J. & Fine, P. A. (2016). The grounded expertise components approach in the novel area of cryptic crossword solving. *Frontiers in psychology*, 7, 1–21.

[Greimas 1967] Greimas, A. J. (1967). L'écriture cruciverbiste. In *To honor roman jakobson: essays on the occasion of his seventieth birthday* (Vol. 1, pp. 799–815). La Haye: Mouton.

[Halliday & Hasan 1976] Halliday, M. A. K. & Hasan, R. (1976). *Cohesion in English*. London: Longman.

[Hardcastle 2007] Hardcastle, D. (2007). Cryptic crossword clues: generating text with a hidden meaning. In *Proceedings of the eleventh european workshop on natural language generation* (pp. 147–150). Association for Computational Linguistics.

[Kilgarriff 2004] Kilgarriff, A., Rychlý, P., Smr, P., & Tugwell, D. (2004). Itri-04-08 the sketch engine. *Information Technology*.

[Lewis 2006] Lewis, M. B. (2006). Eye-witnesses should not do cryptic crosswords prior to identity parades. *Perception*, 35(10), 1433–1436.

[Littman 2000] Littman, M. L. (2000). Computer language games. In T. Marsland & I. Frank (Eds.), *International conference on computers and games* (pp. 396–404). Lecture notes in computer science. Berlin: Springer.

[Moxley 2015] Moxley, J. H., Ericsson, K. A., Scheiner, A., & Tuffiash, M. (2015). The effects of experience and disuse on crossword solving. *Applied Cognitive Psychology*, 29, 73–80.

[Nickerson 2011] Nickerson, R. S. (2011). Five down, absquatulated: crossword puzzle clues to how the mind works. *Psychonomic bulletin & review*, 18, 217–241.

[Sketch Engine] *English TreeTagger PoS tagset with Sketch Engine modifications*. (n.d.). Retrieved from https://www.sketchengine.eu/english-treetagger-pipeline-2/

[Pham 2016] Pham, T. (2016). The register of English crossword puzzles: studies in intertextuality. In C. Schubert & C. Sanchez-Stockhammer (Eds.), *Variational text linguistics: revisiting register in English* (pp. 111–136). Berlin: De Gruyter Mouton.

[Rambousek 2004] Rambousek, J. (2004). Between language play and language game. *Theory and Practice in English Studies*, 2, 165–171.

[Smith 1986] Smith, G. & Du Boulay, J. (1986). The generational of cryptic crossword clues. *The Computer Journal*, 29(3), 282–284.

[Stratmann 1978] Stratmann, G. (1978). Versuch uber die Kunst des britischen Crossword und Appell, dieselbe zu importieren (nebst einem eingedeutschten Exempel). *anglistik und englischunterricht*, 4, 125–136.

[Underwood 1994] Underwood, G., Deihim, C., & Batt, V. (1994). Expert performance in solving word puzzles: from retrieval cues to crossword clues. *Applied Cognitive Psychology*, 8(6), 531–548.

[Vântu 1991] Vântu, I. (1991). La sémiotique des mots croisés. Revue roumaine de linguistique, 23–29.

[Williams 1979] Williams, P. W. & Woodhead, D. (1979). Computer assisted analysis of cryptic crosswords. *The Computer Journal*, 22(1), 67–70.

Modern amendment drafting - The road to XML

Bert Willems, FontoXML

Abstract

This paper reports on an experiment to build a system which aids in the drafting of amendment documents. The system provides a mechanism to help validate the correctness of amendments. Furthermore, the system is able to semi-automatically sort the amendments in voting order and simulate the effects of amendments on the law. The proposed implementation is based on XML technology, an XML editor and machine learning.

1. Context

Disclaimer: The following is a simplified representation of reality intended to provide just enough context to the reader of this paper. It is not intended as a comprehensive introduction to legislation and parliamentary procedures.

New legislation is often introduced by augmenting an existing law rather than introducing an entirely new law. This process is called "amending" the law. An amendment is a formal or official change made to a law, contract, constitution, or other legal document. It is based on the verb to amend, which means: to change. Amendments can add, remove, or update parts of these legal documents.

For example:

3.9 In paragraph A57C(b)(vi), for "his", substitute "their".

This amendment replaces the word "his" with "their" in paragraph A57C(b)(vi).

The process of making a change to the law starts with a deputy (a member of the parliamentary commission and/or the parliament) writing an initial draft of the amendment. This initial draft is sent to the drafting office. The drafting office's job is to ensure that each legal document (including amendments) is well written, unambiguous and consistent with the other legal document(s) in the legal system. It is often the case that they make changes to the draft or they need to make changes to other parts of the law in order to keep the legal system consistent. This often requires discussion with the deputy and new versions of the draft go back and forth.

The current process in the parliaments (at least in Italy and the Netherlands) is still very manual: sometimes the amendments are written on paper which needs to be digitalized first. Other times the amendments are written in Microsoft Word. In both cases, the amendments need to be converted before they can be published. The amendments are often published as PDF files or as HTML on a website.

Both the Dutch and the Italian parliaments use digital workflows and tools, for example, to register amendments. The content of those systems is still primarily Word-based. Adopting an XML-first workflow would eliminate the need for conversion and would enable a whole new suite of tools. However, introducing changes to parliamentary processes is a time-consuming task, often requiring a change in the statute.

2. Introduction

This paper reports on an experiment testing whether it is possible to create an XML authoring system which pro-actively aids during the drafting lifecycle of amendment documents, from initial draft to voting. The goal of the system is to provide a user interface (UI) that is quite similar to Microsoft Word but it is XML based. Having an XML-first workflow reduces the conversion burden but that is not something the authors of amendment documents are concerned with. Therefore, in order to get users to adopt the new system, it must provide clear benefits to those users.

The benefits realized during the experiment are:

1. Assisted checking whether an amendment is correct.
2. Ordering amendments for voting.
3. Simulation of the effect of an amendment to the actual law.

The scope of the experiment is to test whether it is possible to realize those benefits in a software system. The first part of the system is a UI where users write the content of an amendment document. The UI is implemented using a WYSIWYG XML editor. The second part of the system provides real-time warnings and on-demand simulation in case there are no warnings.

2.1. Validating amendments

Although the final judgement of whether an amendment is considered valid and correct must be left to humans at all times, the system assists where it can. For example, if a user writes an amendment to change article X in law Y, the system can test whether article X actually exists in law Y and warn the user if it doesn't. The system can also determine whether the basic components (location and action) of an amendment are actually present and give a warning in case they're missing.

2.2. Ordering amendments

The goal of ordering amendments is to reduce the amount of voting that needs to happen by considering amendments that have a bigger impact first. The ordering of the amendments is rule-based. The rules depend on the location of the target of the amendment and the amendment action.

Consider the following (abstract) rule set:

1. Amendments are first grouped and sorted by article.
2. Within each sorted group, sort the amendments from those with the biggest impact to the ones with the smallest impact.

For example:

Consider amendments A1 and A2. A1 proposes to change a single word in article 1 and A2 proposes to delete article 1. Since the impact of A2 is much bigger than A1, and in fact "contains" A1, it should be first in the list.

The system should warn the user in case the ordering in the amendment document is different from the ordering prescribed by the rule set.

2.3. Simulation

The goal of the simulation is to *simulate the effect* of the change described in an amendment to *the original text*. In order to do so, the system needs to be able to interpret the amendment and apply it to the original text.

Via this simulation, the user gets a real-time preview of the result of an amendment. This enables him/her to see whether the intended effect is actually achieved.

3. Problem definitions

The system is modeled around four main problems:

3.1. Problem 1: Segment amendments in an amendments document

An amendment document may contain multiple amendments. The system must be able to recognize the start & end position of each individual amendment because subsequent problems work on the individual amendment level.

3.2. Problem 2: Recognize the location, action and operand information

Each amendment essentially describes a change to the original text. For the next problems, it is required for the system to understand the nature of the change. Essentially, it needs a model of each change.

The model requires:

1. Location information; Where does the change need to be made in the original text.
2. Action information; How does the original text need to be modified.
3. Operand information; Depending on the action information.

This is essentially an information extraction problem.

3.3. Problem 3: Order the amendments according to the rule set

The ordering algorithm is defined as a rule set that takes the model recognized by Problem 2 [39] as an input. Essentially this problem can be represented as a topological sort with constraints.

3.4. Problem 4: Generate the simulation

The simulation can be modelled as a transformation on the original text taking the result of Problem 2 [39] as an input for the transformation.

Subproblems:

1. Resolve the original text being modified.
2. Find the location in the original text based on the location information.
3. Apply the action and operands.
4. Create a rendition of the result.

4. Implementation

The architecture of the system is based on the Apache UIMA architecture [Lally and Ferrucci 2004]. It is a component software architecture for the development of analytics for the analysis of unstructured information. This allows the problems introduced in the previous section to be decomposed into sub-problems, each which its own solution forming a system that can be quickly adapted to other problems by swapping analytic components in and out. It essentially forms a pipeline through which information flows. Each stage adds additional information in the form of annotations. The implementation takes XML as an input instead of unstructured text. The annotations can be on both the character level, DOM level as well as on both levels simultaneously.

An example of an analysis pipeline would be to use Regular Expressions to extract information which is then fed into a second stage where a machine learning algorithm could take the matches as input features.

The following sections describe how each previously listed problem is solved.

4.1. Segmenting amendments (Problem #1)

Any given amendment document is likely to contain multiple amendments. Sometimes on the same law but is not uncommon to update multiple laws with one set of amendments. Think of an update of multiple laws as a database transaction happening on multiple tables; you want to preserve the referential integrity (or the integrity of the legal system in this case).

The goal of this stage in the pipeline is the recognize where each individual amendment starts and where it ends. The result of the segmentation can be stored in the XML document resulting in a richer XML structure.

Looking at this problem closer, it is composed of two tasks:

1. Distinguish between the preamble, postamble and the body (containing the amendments we care about).
2. Distinguish between individual amendments which are often grouped in articles.

We treat these problems as a sequence classification problem. We trained 2 models implemented using the Conditional Random Fields (CRF) algorithm [Lafferty, Andrew and Fernando 2001]. According to the research performed by Fuchun Peng and Andrew McCallum [Peng and Andrew 2004], CRFs work well for extracting structured information from research papers, achieving good performance. Although amendments are different from research papers, both are generally well-structured and some of the tasks, including segmentation overlap. This algorithm is also used in similar implementations like GROBID [GROBID 2008-2017] and MALLET [McCallum 2002].

4.1.1. Linear Chain Conditional Random Fields (CRF)

We'll treat some of the problems as sequence classification problems. We'll use a well-known algorithm called Conditional Random Fields (CRFs) to solve these problems.

According to Wikipedia:

> CRFs are a class of statistical modeling method often applied in pattern recognition and machine learning and used for structured prediction. CRFs fall into the sequence modeling family. Whereas a discrete classifier predicts a label for a single sample without considering "neighboring" samples, a CRF can take context into account; e.g., the linear chain CRF (which is popular in natural language processing) predicts sequences of labels for sequences of input samples.

> CRFs are a type of discriminative undirected probabilistic graphical model. It is used to encode known relationships between observations and construct consistent interpretations. It is often used for labeling or parsing of sequential data, such as natural language processing or biological sequences and in computer vision. Specifically, CRFs find applications in POS Tagging, shallow parsing, named entity recognition, gene finding and peptide critical functional region finding, among other tasks, being an alternative to the related hidden Markov models (HMMs).

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

CRFs take into account the context in which a word appears and some other features like "is the text made up out of numbers?". More precisely: an input sequence of observed variables X represents a sequence of observations (the words

with the associated features which make up a sentence) and *Y* represents a hidden (or unknown) state variable that needs to be inferred given the observations (the labels). The Y_i are structured to form a chain, with an edge between each $Y_{(i-1)}$ and Y_i. As well as having a simple interpretation of the Y_i as "labels" for each element in the input sequence, this layout admits efficient algorithms for:

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

For a more detailed introduction to CRFs, see An Introduction to Conditional Random Fields for Relational Learning [Sutton and McCallum 2012].

For the implementation of the CRFs, an implementation based on CRFSharp [Fu 2017] is used. CRFSharp is a .NET Framework 4.0 implementation of Conditional Random Fields written in C#. Its main algorithm is similar to CRF++ written by Taku Kudo [Kudo 2017]. It encodes model parameters by L-BFGS. Moreover, it has many significant improvements over CRF++, such as totally parallel encoding and optimized memory usage. The CRFSharp implementation was modified to target the .NET Standard 2.0 to allow cross-platform usage in .NET Core applications.

4.1.2. Model description

The segmentation model is split into two smaller models. The first model segments the amendment text from the surrounding text like introductory and closing words. The second model segments individual articles and the amendments within them.

The most notable extracted features are:

1. Whether the text is inside a heading;
2. Whether the text starts with a list marker, e.g.: 1., a., 1), etc.;
3. Whether the sentence matches a Regular Expression testing for article titles.

The tags used are:

1. preamble
2. heading
3. paragraph
4. amendments
5. article
6. clause (clause of an article, can be nested)
7. postamble

The first model is evaluated on the entire text of the amendments documents. The result of the first segmentation is then fed into the evaluation of the second model which segments the individual articles and amendments within them.

Both models are trained using Conditional Random Fields on a training. The corpus size of the text segmentation model is 35 documents. The corpus size of the amendments segmentation model is 40 documents. Both corpora are drawn from published Dutch amendment documents and tagged by hand. Both models were trained with a maximum iteration count of 1000 using L2 regularization running on 8 threads in parallel. The training of the text segmentation model took 4 minutes while training the article model takes around 6 minutes on a laptop with an Intel i7-4702HQ processor and 16GB of RAM. Training was clearly CPU bound; the 8 logical processors were 100% utilized. Memory usage was around 1.5 GB. Training speed can possibly be improved using the GPU rather than the CPU. However, this was not explored.

The trained models are evaluated against previously unseen examples. Both models are scored on the overall performance of all their labels. Both models are scored using the accuracy and F_1 metrics which are common scoring metrics.

Table 1. Evaluation results

Model name	Accuracy	F_1
Text segmentation	99,53%	99,72%
Amendment segmentation	95,36%	95,31%

As seen in Table 1 [41], the models are quite accurate. This is mostly due to the predictable and precise nature of the amendment documents.

The result of the evaluation of both models is fed back to the UI where the user sees a suggestion to apply markup an entire article by pressing one button. This UI allows the user to make corrections in case the model did not predict the correct structure.

4.2. Recognizing location, action and operand information (Problem #2)

For each amendment, the location, action and operation information must be extracted from the raw amendment text so the system can reason about it. The location information contained details on where the change should be effectuated with respect to the law(s) being modified.

This information includes for example:

- The name of the law;
- The number of the article;
- The number of the clause;
- The position in the text;

The action and operand information are related. The action describes the type of modification. For example, a replacement of a word in the text or the insertion of a new clause. The required operand information depends on the action. For example, a word replacement requires the string to match and the string to replace each match with. A deletion of a clause does not require any further info.

The implementation uses the analysis pipeline architecture again in order to divide the problem into sub-problems. The first stage of the pipeline uses lexicons to detect names of laws and common action words like "substitute".

The second stage of this analysis pipeline is a modified version of Stanford TokensRegex [Chang and Manning 2014]. TokensRegex is a generic framework for defining patterns over text (sequences of tokens) and mapping it to semantic annotations. TokensRegex emphasizes describing the text as a sequence of tokens (words, punctuation marks, etc.), which may have additional annotations, and writing patterns over those tokens, rather than working at the character level, as with standard regular expression packages.

An example of a rule would be:

```
'for' '"' (:<left-op>[]+) '"' ',' [have(app:action)] '"' (:<right-op>[]+) '"'
```

This pattern matches the string: *for "his", substitute "their"*. The pattern consists of literal tokens, 'for' and '"' which must be matched exactly. The pattern also contains named capture groups (left-op and right-op) to extract the operands. Finally, the pattern contains a token which must have an app:action annotation. The action annotation is set in the previous stage of the pipeline using a lexicon. The extraction rules were written by hand.

A notable extension made to the TokensRegex syntax is to also allow XPath expressions to be used within the expression. This extension allows rules which take the XML tagging into account. This feature is extensively used to allow users to manually tag something that has not been recognized automatically. Consider the following expression:

```
(:<left-op>'"' []+ '"' | [matches-xpath('ancestor-or-self::operand')]+)
```

This expression would match one or more token which make up either a quoted string -or- tokens which are wrapped in an `<operand/>` element.

The third stage of this analysis pipeline disambiguates and combines the extracted annotations into an amendment graph representing all the extracted information of each individual amendment. This graph represents both the location information as well as the action information. It is convenient to combine both types of information into a single graph because some of the information overlaps. For example, the word to replace is both location information (which word) as well as an action operand.
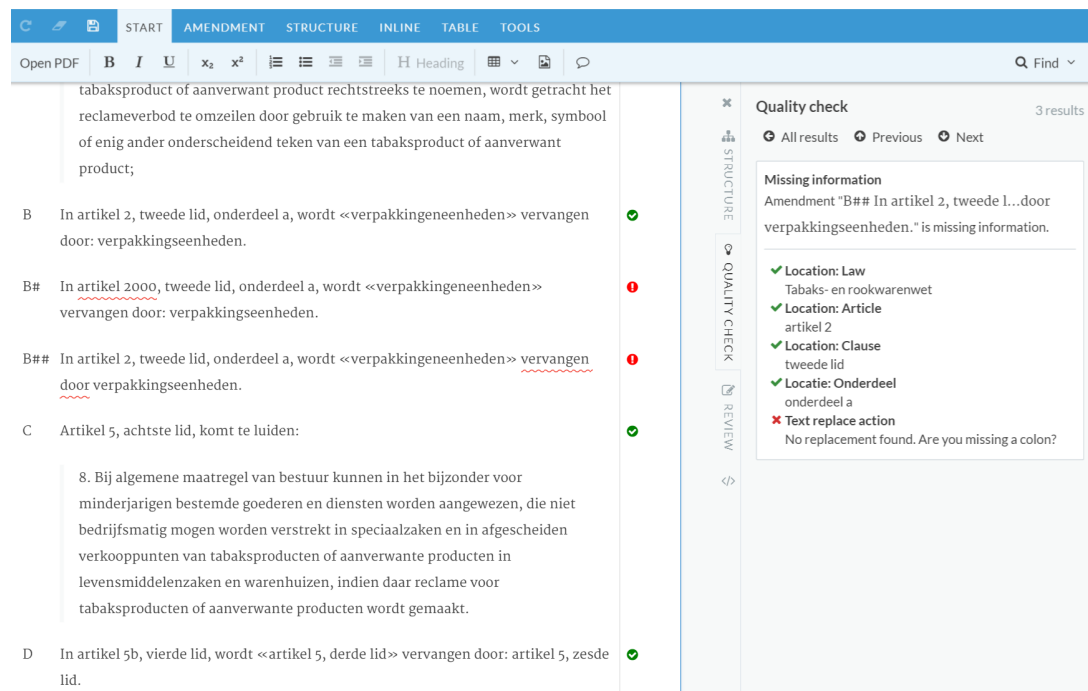
The final stage validates the constructed amendment graph. It for example, tests whether the target does actually exist in law being amended. It also tests the amendment for having at least one target and one action. More details of the location validation are provided in the section Simulating the effect (Problem #4) [43].

4.3. Validating amendments

In section Recognizing location, action and operand information (Problem #2) [42] it is shown how a graph is constructed and validated. The result of that validation is shown in the UI as well as the graph in the form of a table of extracted information.

Figure 1. Validation warnings

See Figure 1 [43]: on the left-hand side you'll see the amendment document. The amendment in clause B is valid, meaning all the location- and action information was extracted and validated successfully. Clause B# is invalid because article 2000 does not exist in the target law. This is signaled by the red squiggle underline. Clause B## is invalid because it is missing a colon after the words "vervangen door". On the right-hand side you'll see the details of the warning for clause B##.

4.4. Ordering the amendments (Problem #3)

Warning: this section has not been implemented and presents only the conceptual idea of the implementation.

Ordering amendments in voting order is a time-consuming task. The order is defined by a set of rules. This problem can be solved with a topological sort with constraints. The conceptual algorithm is straightforward:

1. Gather all amendments graphs extracted from the amendment document.
2. Create a fictitious root node in the merged graph G_m.
3. For each graph G_a in the gathered amendments graphs:
    a. Add to- or reuse in G_m the location nodes in G_m for each location part in G_a.
    b. Add to- or reuse in G_m the action node.
    c. Add the amendment node to G_m.
4. Assign weights to all the location and action nodes.
5. Sort the amendment nodes based on the weight of the shortest path to the root node.

Once the amendment nodes are sorted it is easy to create the ordered list as the system thinks it should be. The system can then use that ordered list to determine whether it differs from the current ordering in the amendment document and warn the user if they differ.

The tricky part of such an implementation is not the algorithm: it is understanding the ordering rules and defining a weighting scheme based on those rules. These ordering rules are different for each legal system. It may very well be that for some systems it may not be possible to express the rules as a weighting scheme. In all cases, the final order must be left to the user.

4.5. Simulating the effect (Problem #4)

Simulating the effect of an amendment to the original text of the law may help users to understand its impact. This is essentially a transformation problem where the transformation is generated from the extracted information in the amendment.

The system performs the following steps accomplish this task:

1. Retrieve the original law.
2. Generate a transformation.
3. Apply the transformation.
4. Show the result to the user.

The first step is to retrieve the original law. Fortunately, the Dutch government publishes the laws as XML documents on the web. For the purpose of this experiment, we downloaded a couple of laws and we implemented a simple web-service to retrieve the XML content of a law based on its name. The name of the law is available in the amendment graph as described in section Recognizing location, action and operand information (Problem #2) [42]. This service does not yet take time into account: in the Dutch legal system, laws change over time due to amendments becoming effective on certain dates. Figuring out the effective law is a subject of its own and is out of scope for this paper.

The second step is to generate the transformation. Essentially this step is generating an XSLT stylesheet with a single template. The templates match attribute is generated by creating an XPath expression based on the location information in the amendment graph. The body of the template is generated based on the action and operand information in the same graph. In this step, the amendment graph acts as an Abstract Syntax Tree (AST) which is compiled to XSLT.

The third step is as easy as applying the generated XSLT stylesheet on the retrieved law document.

The final step is to create a rendition of the modified law document. This rendition includes change highlighting to make it easier for a user to see what will be changed.

As seen in Figure 2 [44], The UI highlights the changes using familiar change representation: text in red and strikethrough is deleted and green underlined text is added. The yellow dotted border indicates the area affected by the simulated amendment.

Figure 2. Simulation result
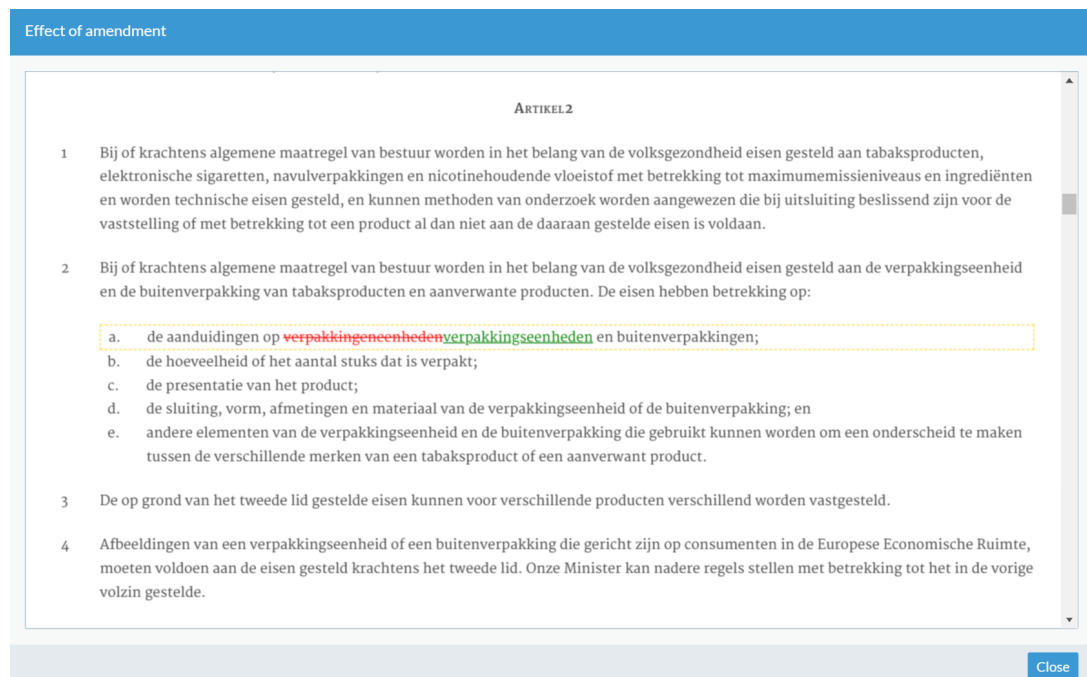
The system does not allow the simulation to be stored and it makes it clear to the user that it is a simulation holding no legal value.

5. Human in the loop

Actual field testing has not been performed at this stage. However, it is already clear that the system will not be accurate in all cases. This is because human language is ambiguous, even in the case of amendments where language is tightly controlled.

In order to not come to a grinding halt, the system is designed to keep the human-in-the-loop (HITL). This effectively means the user and the system work together to accomplish the tasks. The system provides a UI through an XML editor.

If, for example, an operand seems to be missing (e.g. not recognized by the system), the user is provided with a warning. The user can determine whether the required information was indeed missing or whether the information was present but not recognized. In the latter case, the user can manually label the operand by wrapping it in the appropriate XML tag.

6. Conclusions and further work

The experiment shows that it is possible to build a system that can aid in the drafting lifecycle of amendments using some relatively straight-forward techniques. The system includes a mechanism to validate the correctness of an amendment from a technical perspective. Furthermore, the system aids in the ordering of amendments for voting. Finally, the system provides a method for simulating the effect of an amendment on the actual law.

The main problem is reliable information extraction. Although information extraction algorithms are getting more powerful, there is still noise due to the nature of human language. With XML authoring a hybrid approach can be created where users disambiguate during the authoring process by simply applying markup in case the system gets it wrong. With the addition of XPath to the TokensRegex regular expression language, it is possible to write rules that take manual disambiguation into account.

The described Apache UIMA inspired analysis pipeline architecture works well for this kind of information extraction problems. An observation to make is the described system is essentially a compiler for human language: It takes in the text of an amendment which is essentially lexing and tokenizing. From the tokens, an amendment graph is constructed which is an Abstract Syntax Tree (AST). From the AST an XSLT stylesheet is generated which outputs the modified law. Compiler errors are shown to the user as warnings providing hints that something is not correct just yet.

From a technical perspective, the next step would be to be able to automatically learn patterns. This would make the system scalable across different legal systems without the need of rewriting all the rules from scratch every time. The idea is to augment systems like RAPIER [Califf and Mooney 1997] or WHISK [Soderland 1999] to work with the XML-aware TokensRegex.

So far, we approached this experiment from a technical perspective, although it is based on actual issues encountered in the parliaments today. The next step is to take such a system in production. For a legal system, this would mean writing more rules to deal with edge cases. The biggest hurdle are the legal implications though: some of the features described in this paper may not be used legally in some legal systems due to rules of the parliamentary procedures.

Bibliography

[Califf and Mooney 1997] Califf, Mary Elaine, and Raymond J. Mooney. 1997. "Relational Learning of Pattern-Match Rules for Information Extraction." CoNLL.

[Chang and Manning 2014] Chang, Angel X., and Christopher D. Manning. 2014. TokensRegex: Defining cascaded regular expressions over tokens. Stanford University Technical Report, Department of Computer Science, Stanford University. https://nlp.stanford.edu/pubs/tokensregex-tr-2014.pdf.

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

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

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

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

[Lally and Ferrucci 2004] Lally, Addam, and David A. Ferrucci. 2004. "UIMA: an architectural approach to unstructured information processing in the corporate research environment." Natural Language Engineering 10: 327-348.

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

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

[Soderland 1999] Soderland, Stephen. 1999. "Learning Information Extraction Rules for Semi-Structured and Free Text." Machine Learning 34: 233-272.

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

Introduction to CSS for Paged Media

Tony Graham, Antenna House, Inc.

Abstract

CSS can be used for making pages as well as for styling websites. Many people are familiar with CSS in the browser: some are very familiar, but others, not so much. Fewer people, however, are as familiar with using CSS for paged media.

This paper introduces some of the key features of CSS for paged media, including: differences between screen and paged media; sizing and selecting pages; headers and footers; PDF output and PDF accessibility features; printing colour; and numbering pages. This material is adapted from a more comprehensive tutorial that is available online [20].

1. Introduction

CSS is widely used in browsers, editors, and other applications. CSS is used not only for web design but also as the stylesheet specification for a wide range of printing applications as well as for electronic paged media delivered as PDF.

CSS (Cascading Style Sheets) Level 1 became a W3C Recommendation in 1996. CSS 2 became a W3C Recommendation in 1998, and CSS 2.1 (Cascading Style Sheets Level 2 Revision 1), in 2011. As of early 2018, CSS 2.2 is currently under development.

CSS post-Level 2 is popularly known as CSS 3, but there will not be a single, monolithic CSS Level 3 specification. CSS beyond CSS Level 2 has been broken into multiple modules that define separate parts of CSS. These modules are numbered individually. The first versions of modules that build on CSS Level 2 are denoted as Level 3, and each may be superseded by a Level 4 version. For example, CSS Color Level 3 replaces several sections of CSS Level 2, and CSS Color Level 4, which is currently in development, will eventually replace CSS Color Level 3. Modules that do not build on CSS Level 2 features start at Level 1: for example, CSS Multi-Column Layout Level 1. There will not be a CSS Level 4 or beyond.

Individual modules are in varying stages of development and varying levels of stability. The stability levels for all W3C specifications range from Working Draft to Recommendation. The CSS Working Group maintains a separate stability categorization for its specifications that ranges from Rewriting and Exploring through to Stable and Completed.

The CSS Working Group compiles yearly snapshots of the current state of CSS at that point in time. CSS Snapshot 2017[1] lists both Recommendation and Candidate Recommendation specifications as comprising the official 2017 definition of CSS (even though the document defines Candidate Recommendation as the test phase of a W3C specification).

CSS 2.1 (and CSS 2.2) provides only minimal support for paged media output, and its page layout features are not powerful enough. CSS 3, although still under development by the W3C, defines many of the features that are necessary for professional quality formatting, including: advanced page layouts; multiple columns; vertical writing; hyphenation; and multilingual character layout. Antenna House Formatter provides additional features for optimal formatting, including: custom-developed MathML 3, CGM, and SVG rendering; baseline grids; PANTONE® spot colours; and additional properties for controlling Japanese layout.

Using CSS in paged media design for XML and HTML is not yet common but its use is expected to increase as the development of CSS 3 progresses. This tutorial aims to make CSS for Paged Media easy to understand.

2. Web and Paged Media

2.1. @media Rule

An @media rule delimits a set of CSS style sheet rules specific to a target medium. Specify @media print for rules specific to paged media and @media screen for rules specific to screen display.[2]

```
@media print {         /* applies to paged media */
  body {
    margin: 0;
    font-size: 10pt }
  }
@media screen {        /* applies to screen display */
  body {
    margin: 10%;
    font-size: 12px }
  }

body {                 /* applies to all media */
  font-family: sans-serif;
}
```

---

[1] https://www.w3.org/TR/css-2017/
[2] Since AH Formatter is print formatting software, it does not apply @media screen rules for the GUI screen but does apply @media print rules.

2.2. Specifying a Print Style Sheet

2.2.1. <style> Element

A <style> element contains style information for the document. In HTML 4.01, <style> may only appear inside <head>. In HTML 5, <style> may also be used in the body of the document.

```
<style type="text/css" media="print">
…
</style>
```

2.2.2. @import Rule

A print-only style sheet can be created in another CSS file by including it with `@import`.

```
@import url("PrintOnly.css") print; /* PrintOnly.css printing */
```

2.2.3. Media attribute of <style> and <link> elements

Specifying print as the media attribute value links the print style sheet with the <style> or <link> element.

```
<link rel="stylesheet" type="text/css" media="print" href="foo.css">
```

2.3. Differences Between Screen and Paged Media

2.3.1. Design approach

Screen display and printing (or any sort of paged media) require different approaches to designing the layout.

The size and aspect ratio of a screen display may change depending on the display environment, so it is hard to know how to accurately specify the size and arrangement of the layout target. The style specification should consider using relative dimensions to accommodate various environments.

In printing, there is an expectation that formatted objects are arranged neatly on fixed-sized pages, therefore, the layout specification should precisely control the layout by specifying absolute dimensions for the size and position of the formatting objects, starting with the size of its characters.

2.3.2. Breaks

Breaks happen in both paged and unpaged documents. For example, text is broken into lines, and the text in a block that has `column-count` greater than 1 may be broken into columns. However, and unsurprisingly, breaks are more common, and more of a concern, when a run of text is also broken across pages. There are properties for forcing or avoiding breaks within or between elements. Additionally, the `widows` and `orphans` properties control the minimum number of lines of text before or after a break in a block of text.

2.3.3. Floats

In unpaged media, a box can float to the left or right. In paged media, it can also float to the top or bottom of the page (and AH Formatter implements more detailed control over floats). Items that you might float include graphics, sidebars, and footnotes.

2.3.4. Navigation

Paged media (i.e., books) have well-developed conventions for navigating between pages.

Pages are typically numbered, and, often, the front matter is numbered in a different style and sequence to the main text.

The page number and, often, the book, chapter or even section title may appear at the periphery of the page. Dictionaries have their own conventions for indicating the first and last entries on each page. CSS defines 16 regions around the edge of the page for presenting this sort of information.

The table of contents (or tables of contents) and index facilitate non-sequential access.

A chapter (or other significant division) may start on a right-hand page (for left-to-right writing mode documents), and the chapter start may have a different appearance to other pages (and possibly different headers and footers).

2.3.5. Left and right pages

A document that is printed on both sides of the page and bound into a book-like form (even a document that is duplex-printed on an office printer and placed in a folder) will form two-page spreads with a left-hand and a right-hand page. Also, because a book is bound, it is easier to see the details near the outer edges of each page as you leaf through the document than it is to see the inner edges of the page near the binding. Thirdly, the sequential reading order of the pages makes it convenient to think of each two-sided leaf of the document as having a 'front' and a 'back' side.

All of these aspects can affect the page design. For example, chapter openings are typically (but not exclusively) on the right-hand of a spread, since that is the 'front' side of a leaf. Page numbers and any other navigation aids on a page are more likely to be on or near the outer edge of each page so they can be seen more easily when leafing through the document. Thinking of the document as a sequence of two-page spreads also raises questions of whether the facing pages should be symmetrical around the gutter and whether items such as graphics can span across the two pages.

2.3.6. The printed book

Several things should be considered for a document that is to be printed rather than only viewed on screen.

There may be constraints on the page size. A document that is meant to be printed by the end user may be sized to suit the paper size of an office printer: Letter size in the USA; A4 in most of the rest of the world; or A4 or JIS-B5 in Japan. A car handbook, on the other hand, is usually a convenient size for a car glove compartment. Trade paperbacks have a range of conventional sizes, and choosing an unconventional page size could affect the sales of a book.

If the paper is not sufficiently opaque, the text on the opposite side of the paper may show through. The effect is made worse if the text on each side of the paper is not aligned.

Figure 1. Effect of show-through with non-aligned and aligned text

Introduction to CSS for Paged Media

Using CSS in paged media design for XML and HTML is not yet common but its use is expected to increase as the development of CSS 3 progresses. This tutorial aims to make CSS for Paged Media easy to understand.

Introduction to CSS for Paged Media

Using CSS in paged media design for XML and HTML is not yet common but its use is expected to increase as the development of CSS 3 progresses. This tutorial aims to make CSS for Paged Media easy to understand.

Graphics, and other design elements, that extend to the edge of the page may need to be printed so they extend past the edge of the page (see Section 3 [52]). If they do not bleed past the edge of the page, then any inaccuracy when trimming the page to its correct size after printing and binding could result in a white strip between the graphic and the edge of the page. Conversely, the graphic should not have significant details close to the edge of page in case the trimming takes off too much rather than too little.

Even the binding method may need to be considered when designing the book. Perfect binding or a wire binding may reduce the visible or usable area of the gutter between pages. If the pages of a book are gathered into signatures and then trimmed, the pages in the middle of the signature may have more trimmed from their fore-edge than is trimmed from the pages on the outside of the signature.
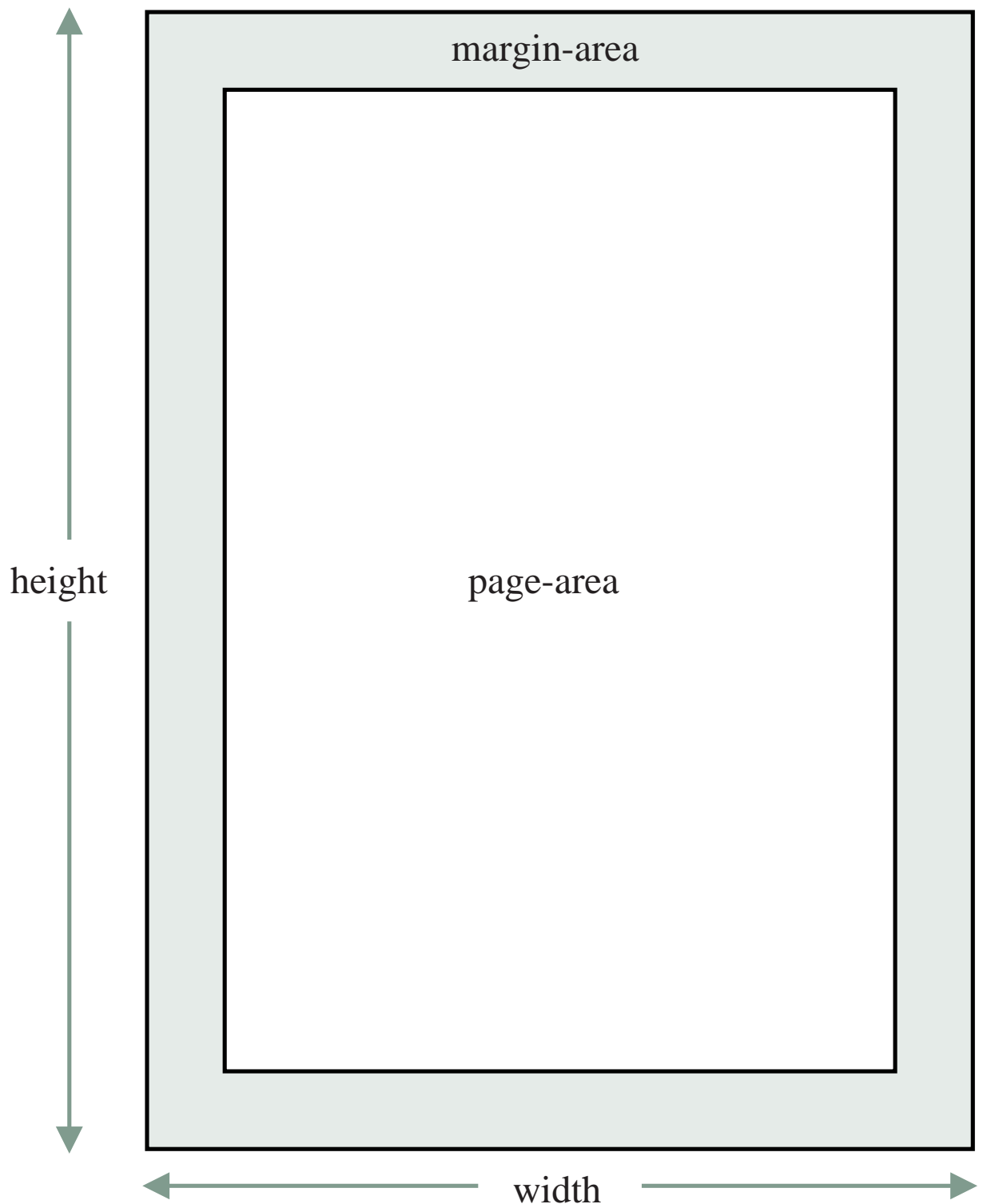
Figure 2. Effect of binding method

Glue

Gutter reduced by perfect binding   Fore-edge reduced by trimming of signature
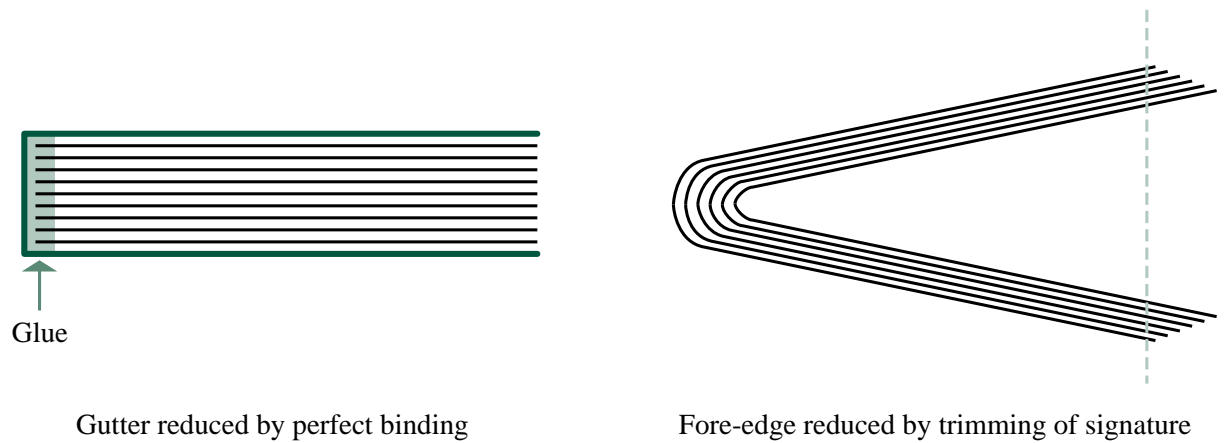
3. Page Setting

In paged media, the document is formatted as one or more page boxes. The content area of a page box is called the page area. The margin area of a page box is used for headers and footers.

Figure 3. Page and Margin Area

3.1. Western Page Design

Western page design tradition places the page area above the center of the page, since the optical center of the page is considered to be above the geometric center. The gutter margin is traditionally narrower than the fore-edge margin. This is to make it easier for the eye to move from one page to the other. The wider fore-edge allows room for the thumbs to hold the page. The two gutter margins, taken together, balance the wider fore-edge margin. The height of the page area matches the width of the page.

Figure 4. Traditional Western page design
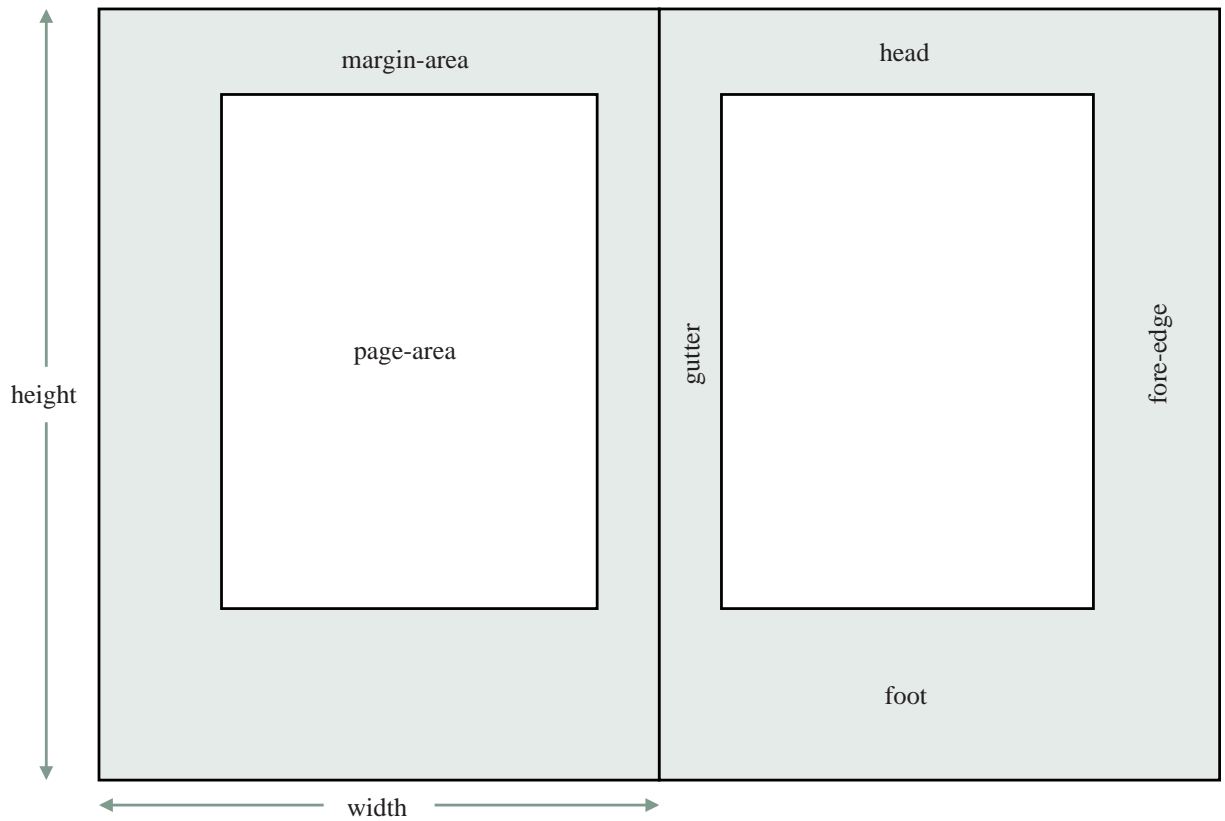
In practice, page designs vary quite a lot. The printed 'page' now includes package inserts for medications, marketing brochures, children's books, parts catalogs, and much more besides. The economics of printing or the need to print on A4 or Letter size paper on an office printer can influence the page design. Asymmetric page design, where the page area has the same position on facing pages, was once a radical idea but now is not uncommon. Furthermore, novels, in particular, are often sold in multiple editions with different page sizes that each reuse the same page areas with reduced margins.

3.2. Japanese Page Design

Japanese can be formatted either with horizontal lines of text and pages progressing from left to right or with vertical lines of text and pages progressing from right to left. Some formatted documents mix the two. Different types of publication are predominately one writing mode or the other: for example, government documentation and educational materials both predominately use horizontal text, whereas novels predominately use vertical text.

For both writing modes, the page area is typically centered on the page media and has proportions similar to the proportions of the media. The best line length is around 52 characters per line for vertical text and around 40 characters per line for horizontal text.

In Japanese text composition, it is common to set the width of the page area in fullwidth characters. Using em for the width property in the @page rule sets the width in characters. Setting the left and right margin values to auto aligns the page area in the center of the page.

```
@page {
 size: A4;
 width: 43em;           /* set width of page area to 43 em */
 margin-top: 30mm;
 margin-bottom: 30mm;
 margin-left: auto;  /* position page area in the center of the page */
 margin-right: auto;
}
```

3.3. @page Rule

An `@page` rule sets basic settings such as page size, margin, page header, and page footer.

```
@page {
  size:   A4;
  margin: 25mm;
  @top-center {
    content: "Sample";
  }
  @bottom-center {
    content: counter(page);
  }
}
```

3.4. Named Page : **page** property

Several types of named `@page` rules can be created, and the `page` property enables switching between them within one document.

```
@page Landscape {   /* "Landscape" named page */
  size: A4 landscape;
}
@page Portrait {    /* "Portrait" named page */
  size: A4;
}
table.WideTable {
  page: Landscape;  /* place a large table on the side of a "Landscape" page */
}
html {
  page: Portrait;   /* Use a "Portrait" page as the default */
}
```

```
<p>Portrait page</p>

<table role="WideTable" border="1" style="width:100%">
<tr>
<td>1</td>
…
<td>18</td>
</tr>
</table>
```

Figure 5. **page** property selects named page
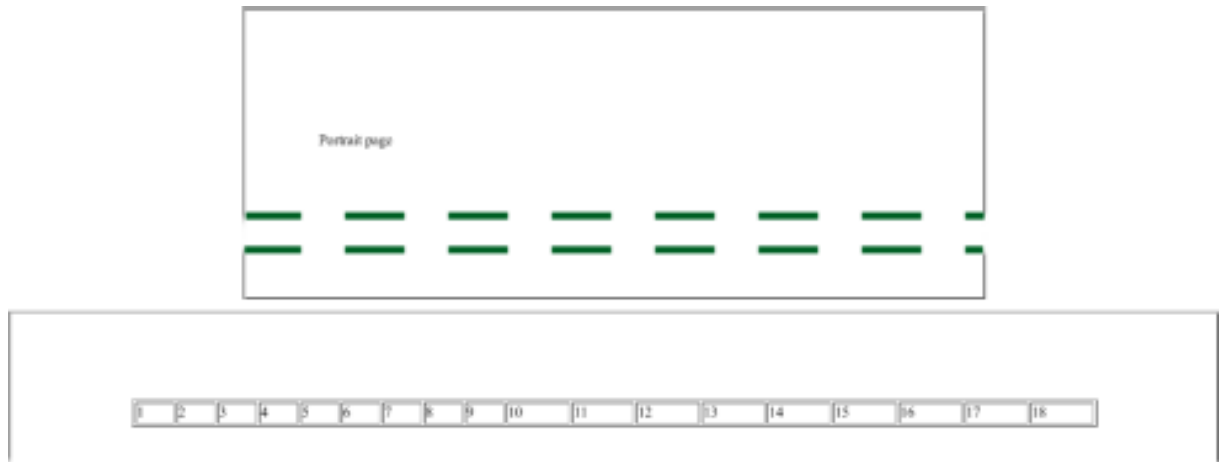
## 3.5. Crop and Registration Marks

An @page rule defines a page box, but the page box may be printed on a page sheet that is larger than the page box. A common reason for this is so images and other content can extend up to the edge of the page box. A physical device such as a printer typically has a non-printable area around the edge of the page sheet where it is not capable of printing reliably, if at all. Printing the page box on a larger page sheet then trimming the page sheet to the size of the page box avoids problems with the non-printable area. Extending images, etc., into the bleed area outside the page box avoids problems if the trimming to size is inaccurate.

Crop and registration marks are printed outside the page box and are used as guides when trimming the page sheet to size and for checking that content printed on both sides of a duplex sheet is aligned correctly. Other information that may be printed outside the page box includes colour bars for checking colour fidelity as well as information identifying the page, its containing document, its version number, etc.

Figure 6. Crop mark terms
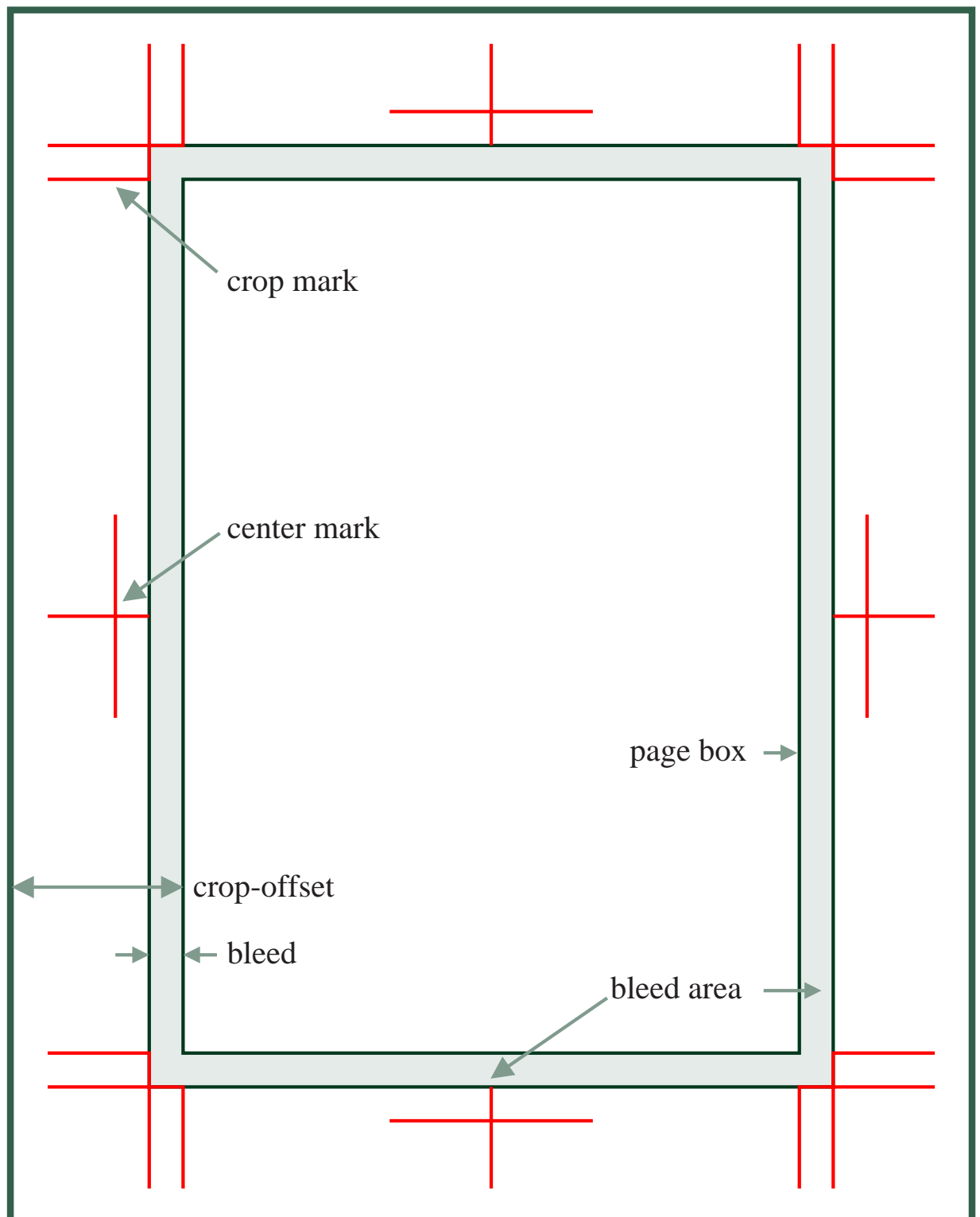
3.5.1. Page bleed area

A graphic, or similar, may *bleed off* (or be *bled-out* from) the cut edge of the page. Extending an image to the edge of the page is often a useful effect. If the image extends just to the edge of the trimmed page, inaccurate trimming could leave a white area along the outer edge of the image. Extending the image past the edge of the page then trimming to size avoids problems from inaccuracy when trimming.

Figure 7. Page bleed

3.5.1.1. Bleed region width : `bleed` property

Specifies the width of the bleed region for trimming. The bleed region extends outwards from the page box. By specifying a negative value to the margin box margin, the range of the block can be extended to the bleed region.

4. Headers and Footers

4.1. Margin Boxes

The page header and page footer are assigned to margin box areas around the page.

Margin boxes are named according to their position around the page, as follows: `@top-left-corner`, `@top-left`, `@top-center`, `@top-right`, `@top-right-corner`, `@left-top`, `@left-middle`, `@left-bottom`, `@right-top`, `@right-middle`, `@right-bottom`, `@bottom-left-corner`, `@bottom-left`, `@bottom-center`, `@bottom-right`, and `@bottom-right-corner`.

Figure 8. Location of each margin box
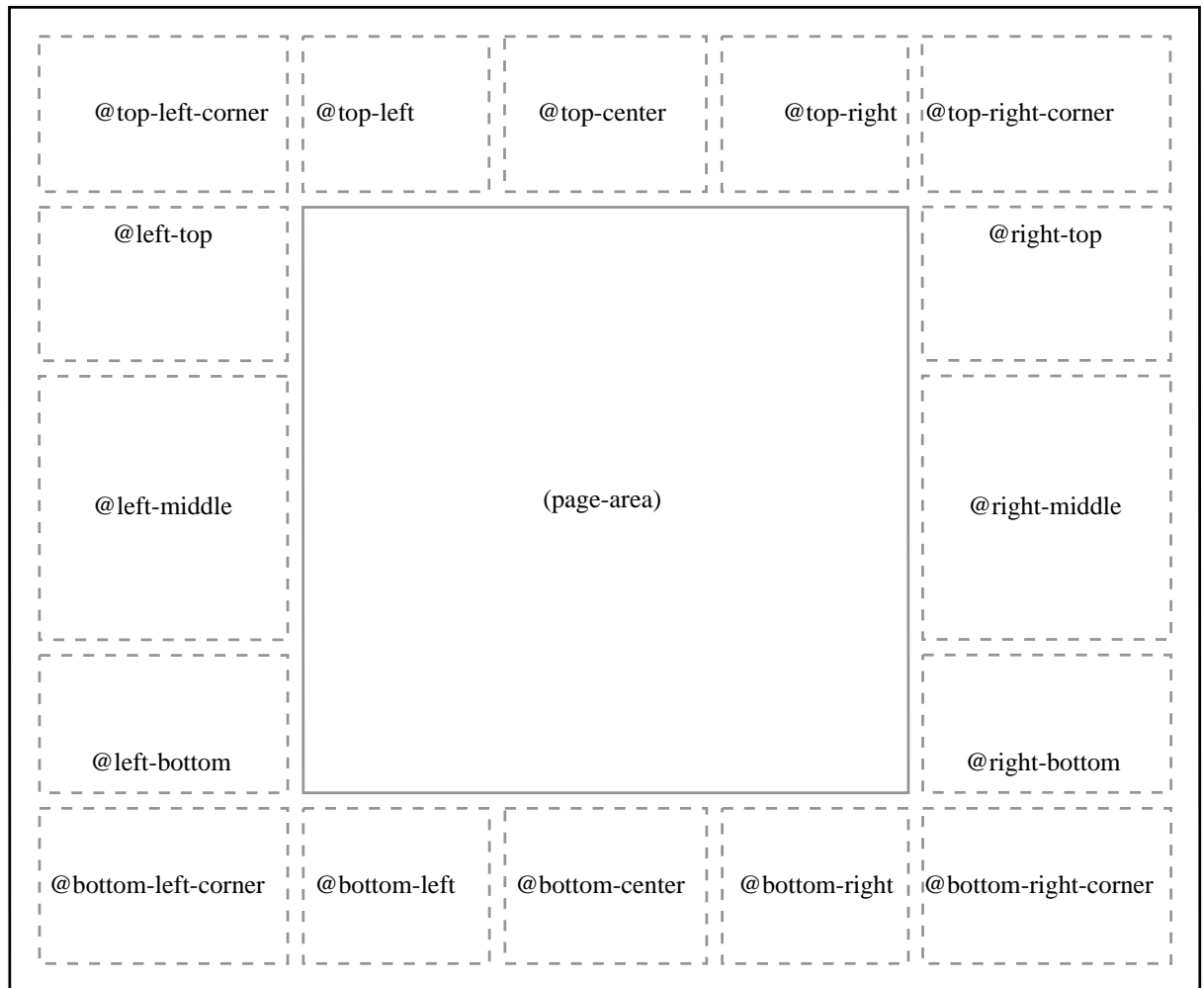

```
@page {
  @top-right {          /* page header */
    content: "Sample";
  }
  @bottom-center {    /* page footer */
    content: counter(page);
  }
}
```

4.2. Running Headers and Page Numbers

4.2.1. Running header setting : **string-set** property and **string()** function

Character strings from the headings in the main body can be displayed in the page header.

```
@page {
  @top-left {
    content: string(Chapter);
  }
}

h1 { string-set: Chapter content(); }
```

4.2.2. Variable strings : **`string-set`** property

Use the `string-set` property to make a named variable for a string.

The string-set value is pairs of a variable name and followed by the content list stored in the named string. Strings defined with a string-set value can be referenced in running headers.

```
h1 {
  /*set contents of h1:before and h1 in Chapter content(before) content()*/
  string-set: Chapter content(before) content();
}
```

4.2.3. **`string()`**

Used to copy the value of a named string into the document.

Strings defined with a `string-set` value are referenced as, for example, `content: string(Chapter);` in running headers.

The required first value is the name of the string.

```
@top-right {    /* Title in right-hand page header. */
  content: string(Chapter);
}
```

If multiple elements on one page each set the value of the same named string, then the named string may have several values on that page. The optional second argument of `string()` specifies which of the possible values to use.

```
@page Index:right {
  @top-left {
    content: string(IndexTerm, first);
  }

  @top-right {
    content: string(IndexTerm, last);
  }
}
```

4.2.4. Move elements to header/footer : **`running()`** position value

Use `position: running(name);` to make an element available for display in a margin box. The name argument is the name by which the element is referred to in `element()` functions.

An element with `position: running(name);` is not shown in its natural place: it is treated as if `display: none;` had been set.

```
p.Title {
    position: running(Title);
    text-indent: 0;
}
```

4.2.5. Insert a running element: **`element()`**

Used to copy a running element into a margin box.

Elements taken out of their natural place using `position: running(name);` are referenced as `content: element(name);` in running headers. A running element can hold one element, including its pseudo-elements and its descendants. Unlike `string()`, `element()` cannot be combined with any other values.

A running element inherits through its normal place in the document.

The required first value is the name of the running element.

```
@top-left {   /* Title in left-hand page header. */
  content: element(Title);
}
```

If multiple elements on one page each set the value of a running element, then the running element may have several values on that page. The optional second argument of `element()` specifies which of the possible values to use.

4.2.6. Page number : counter(page)

`counter(page)` is used for generating page numbers[3].

```
@page {
  @top-right {
    content: "Page " counter(page);
  }
}
```

4.2.7. Total pages : counter(pages)

Total pages can be output together with the current page number.

```
@page {
  @top-right {
    content: "Page " counter(page physical) " of " counter(pages);
  }
}
```

4.3. Left and Right Page Headers: :left and :right

You can set left and right margins as well as headers and footers[4] separately for left and right pages. These may be different again for the first page. You can also hide the titles and page number from the left-hand side of the left pages and the right-hand side of the right pages when it is the first page.

When used together with named pages, the style of the left and right pages and of the first page of each named page can be specified.

```
@page Chapter:left {     /* left page setting */
  margin-left: 23mm;
  margin-right: 27mm;

  @top-left {     /* book title in the running head of the left page */
    content: string(Title);
  }
  @bottom-left {  /* page number */
    content: counter(page);
  }
}

@page Chapter:right {    /* right page setting */
  margin-left: 27mm;
  margin-right: 23mm;

  @top-right {    /* section title in the running head of the right page */
    content: string(Section);
  }
```

---

[3]`counter()` was defined by CSS 2.1, but in CSS 3, a preassigned counter for page numbers is introduced in the page context.
[4]Use `string(Title)` and `string(Chapter)`, see Section 4.2.2 [60].

```
    @bottom-right { /* page number */
      content: counter(page);
    }
}

@page Chapter:first {     /* setting of the first page of a Chapter */
  /* hide page header */
  @top-right { content: none }
  @top-left  { content: none }
}
```

Figure 9. Three pages with different headers and footers

First Page

Right Page

Left Page
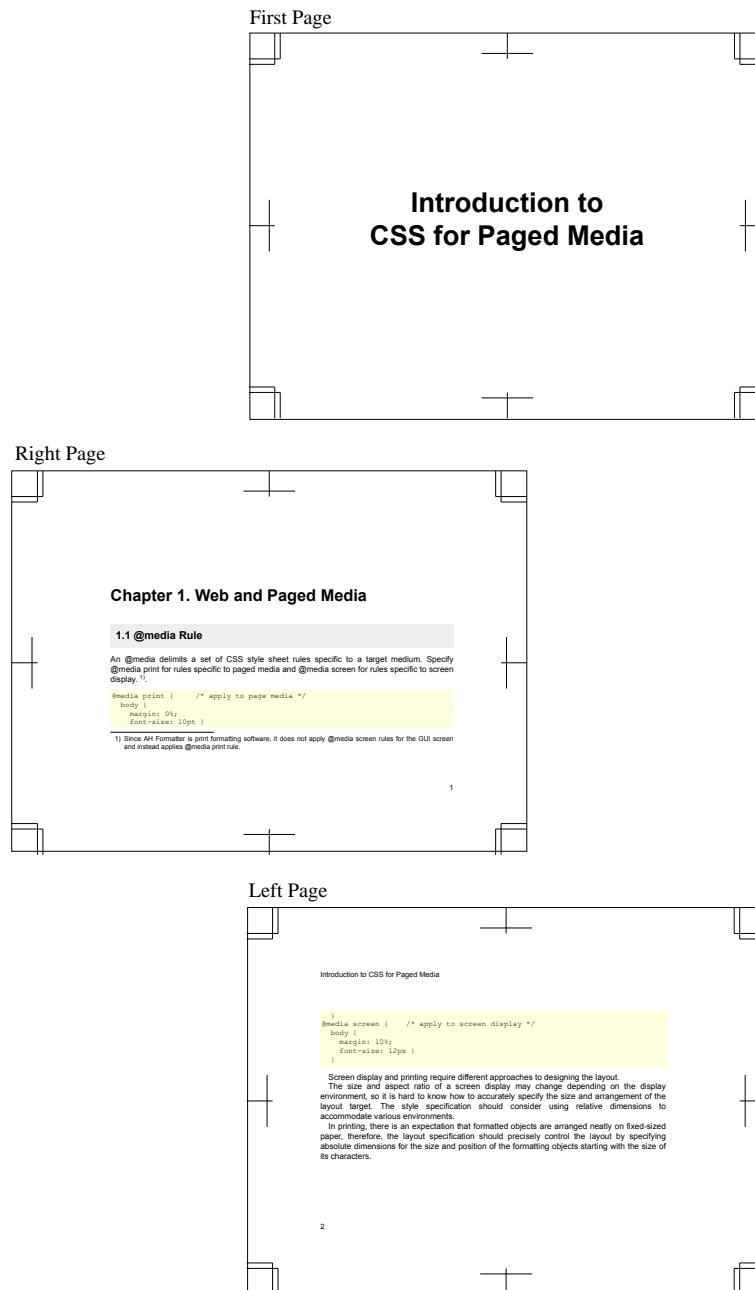
4.4. Last and only pages: `:last` and `:only`

AH Formatter also implements :last and :only pseudo-classes for making page selectors that match on the last and only pages, respectively, of the document.

5. PDF Output

Portable Document Format (PDF) is the most popular ouput format for paged media. AH Formatter provides multiple extensions for taking advantage of PDF features.

5.1. PDF versions

The PDF specification has gone through multiple versions since its first version was published in 1993. Each release has added new features. All non-deprecated features in a PDF version are also included in subsequent versions. A PDF file includes a header identifying the PDF version to which it conforms. A PDF reader will attempt to read any PDF file, even if the file's version is more recent than the version that the reader implements.

Table 1. PDF specification versions

Version	Year	Acrobat Reader version
1.3	2000	4.0
1.4	2001	5.0
1.5	2003	6.0
1.6	2004	7.0
1.7	2008	8
2.0	2017	–

There are also specialized subsets of PDF that have been standardized by ISO. Some of these have multiple versions that are each based on a specific PDF version.[5]

• PDF/X : "PDF for Exchange".

• PDF/A : "PDF for Archiving".

• PDF/E : "PDF for Engineering".

• PDF/VT : "PDF for exchange of variable data and transactional (VT) printing".

• PDF/UA : "PDF for Universal Accessibility".

5.2. 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.

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 are Tagged PDF files that also conform to additional requirements.

AH Formatter embeds PDF tags (StructElem) for HTML/CSS elements and pseudo-elements as shown in the following table:

HTML element	PDF element
html	Document
div	Div
h1	H1
h2	H2
h3	H3
h4	H4
h5	H5
h6	H6

[5]AH Formatter does not generate either PDF/E or PDF/VT.

HTML element	PDF element
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

5.3. 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.[6]

The main features of PDF/UA 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.

5.3.1. Matterhorn Protocol

The Matterhorn Protocol, published by the PDF Association, is a checklist of all the ways that it is possible for a PDF to not conform to PDF/UA. The Matterhorn Protocol document[7] consists of 31 Checkpoints comprised of 136 Failure Conditions. Some failure conditions can be checked programmatically, but others require human review.

---

[6]PDF/UA output is not available with AH Formatter Lite.
[7]https://www.pdfa.org/publication/the-matterhorn-protocol-1-02/

Figure 10. 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

5.3.2. PAC 3 PDF/UA checker

PDF Accessibility Checker 3 (PAC 3)[8] 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. Unfortunately, the program also has some bugs in its checking.

---

[8]http://www.access-for-all.ch/en/pdf-lab/536-pdf-accessibility-checker-pac-3.html

Figure 11. PAC 3 PDF/UA checker

5.4. Document properties

This extension uses custom <meta> elements, for example:

```
<meta name="document-title" content="The document title"/>
<meta name="subject" content="The document subject"/>
<meta name="author" content="The author"/>
<meta name="keywords" content="Comma, separated, keywords, list"/>
…
```

<meta> with the following name values provide information that is stored in the document catalog in the PDF.

Figure 12. Document properties shown by Acrobat Reader
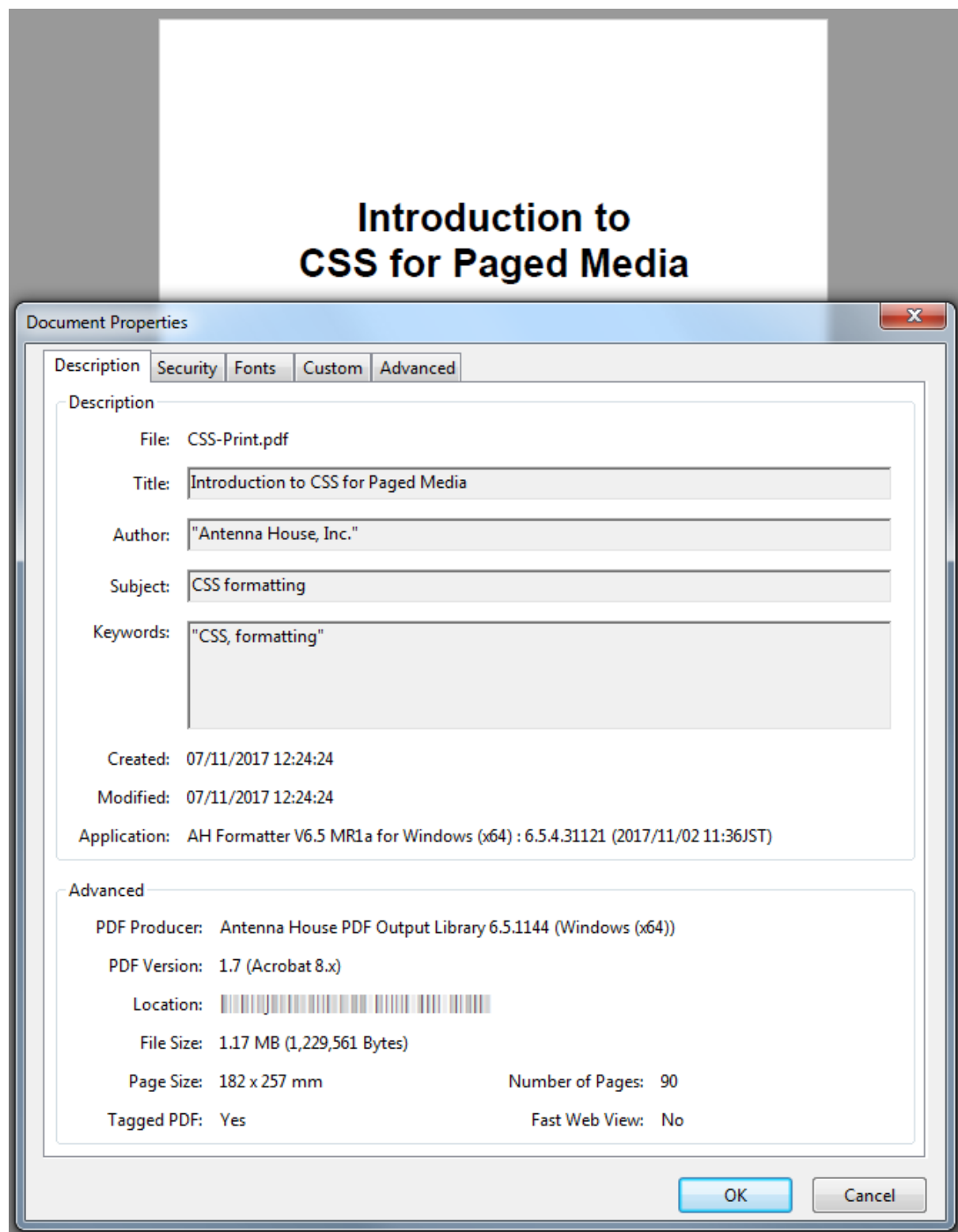
5.4.1. Extensible Metadata Platform (XMP)

XMP is a standard XML format for representing metadata about a file or image. It was originally developed by Adobe, and it is now also ISO 16684-1:2012. Because there is a standard, the XMP, for example, that is embedded in a photo taken by a digital camera can be altered or augmented by the photo-editing program that edits the image. Also, for example, the XMP from every image in a PDF file could be included in the XMP that is embedded in the PDF file.

The XMP standard itself is based, in part, on other metadata standards such as Dublin Core and RDF.

Any XMP file that passes the sanity check implemented in AH Formatter may be embedded in the PDF that AH Formatter generates.

A portion of the XMP extracted from a PDF file is shown below, and the following figure shows the same XMP as presented by Acrobat.

```
<?xpacket begin="" xml:id="W5M0MpCehiHzreSzNTczkc9d"?>
<x:xmpmeta xmlns:x="adobe:ns:meta/">
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
<rdf:Description rdf:about=""
    xmlns:xapMM="http://ns.adobe.com/xap/1.0/mm/">
<xapMM:DocumentID>uuid:4841192B-77AD-0B4B-BD43-8DF3BDE3A5EA</xapMM:DocumentID>
<xapMM:VersionID>1</xapMM:VersionID>
<xapMM:RenditionClass>default</xapMM:RenditionClass>
</rdf:Description>
<rdf:Description rdf:about=""
    xmlns:pdf="http://ns.adobe.com/pdf/1.3/">
    <pdf:Producer>Antenna House PDF Output Library 6.5.1216 (Windows
    (x64))</pdf:Producer>
<pdf:Keywords>"CSS, formatting"</pdf:Keywords>
<pdf:Trapped>False</pdf:Trapped>
</rdf:Description>
<rdf:Description rdf:about=""
    xmlns:xap="http://ns.adobe.com/xap/1.0/">
    <xap:CreatorTool>AH Formatter V6.5 MR3 for Windows (x64) : 6.5.6.31956
    (2018/02/02 12:52JST)</xap:CreatorTool>
<xap:ModifyDate>2018-02-25T16:38:45Z</xap:ModifyDate>
<xap:CreateDate>2018-02-25T16:38:45Z</xap:CreateDate>
</rdf:Description>
…
```

Figure 13. XMP properties viewed in Acrobat
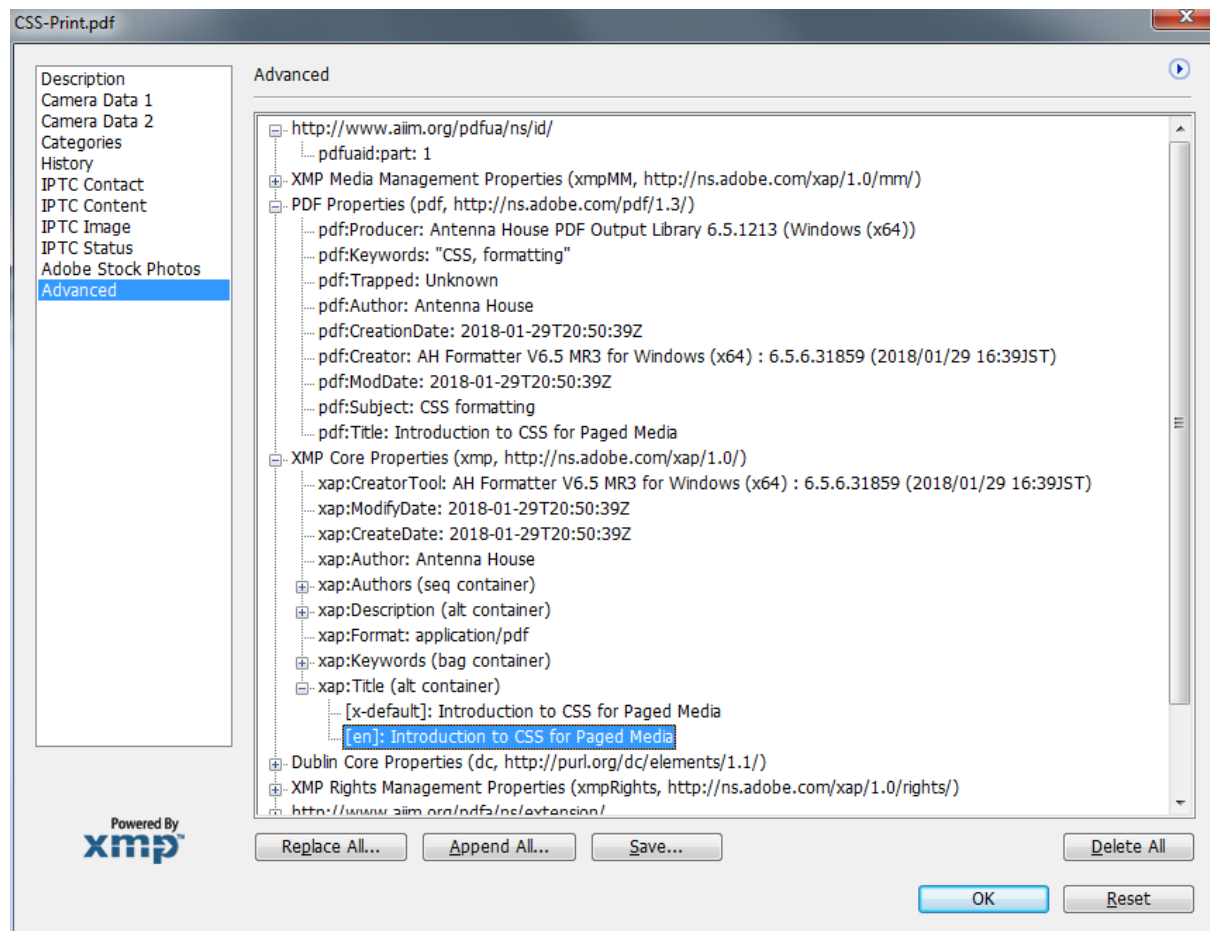
6. Colour Specification

6.1. Printing colour

When you use CSS with your browser, the colours that you see are from the display emitting light. When the red, green, and blue primary colours combine to make a colour, they are additive: the more of each component there is, the lighter the colour. Which is why #FFF is white (or why white is #FFF, depending on your perspective).

Conversely, when you print, the colours that you see are from the light that is reflected from the printed surface. When the cyan, magenta, and yellow primaries combine to make a colour, they are subtractive: the more of each component there is, the darker the colour. `cmyk(1, 1, 1, 0)` should give black, but in practice, it's closer to a muddy brown. That's one reason why black is added as the fourth colour.[9] Using black ink is also less expensive than using a triple quantity of coloured inks. Text is typically printed solely in black to avoid problems if the other three inks are not perfectly aligned. A 'richer' black, which might be used for example in a graphic, can be made by applying solid black over one or more other colours.

---

[9] The 'K' in 'CMYK' is from the black printing plate being the 'key' plate against which the other plates are aligned.

Figure 14. RGB and CMY
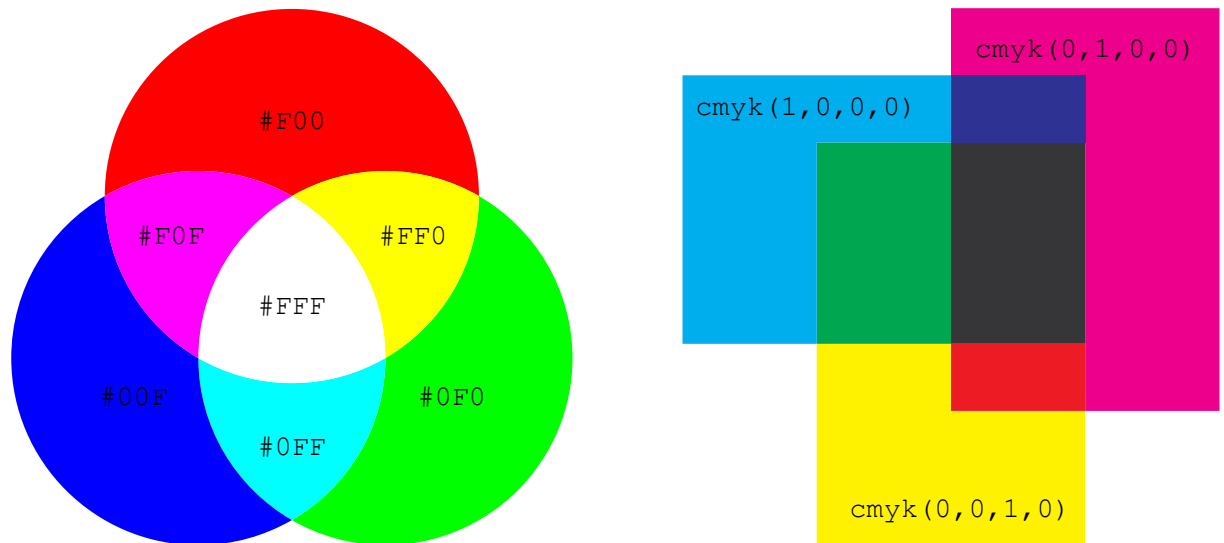
The relationship between RGB and CMY was first demonstrated by James Clerk Maxwell around 1860. Put simply, the printing primaries are the secondary colours of the transmitted light primaries, and vice-versa: cyan is blue plus green, or white minus red, and similarly for magenta and yellow. However, as the following figure shows, CMYK printing inks do not have the same gamut (i.e., colour range) as the sRGB colour space that is used for RGB colours on the web, and neither covers the full gamut of visible light.

Figure 15. sRGB and CMYK gamuts
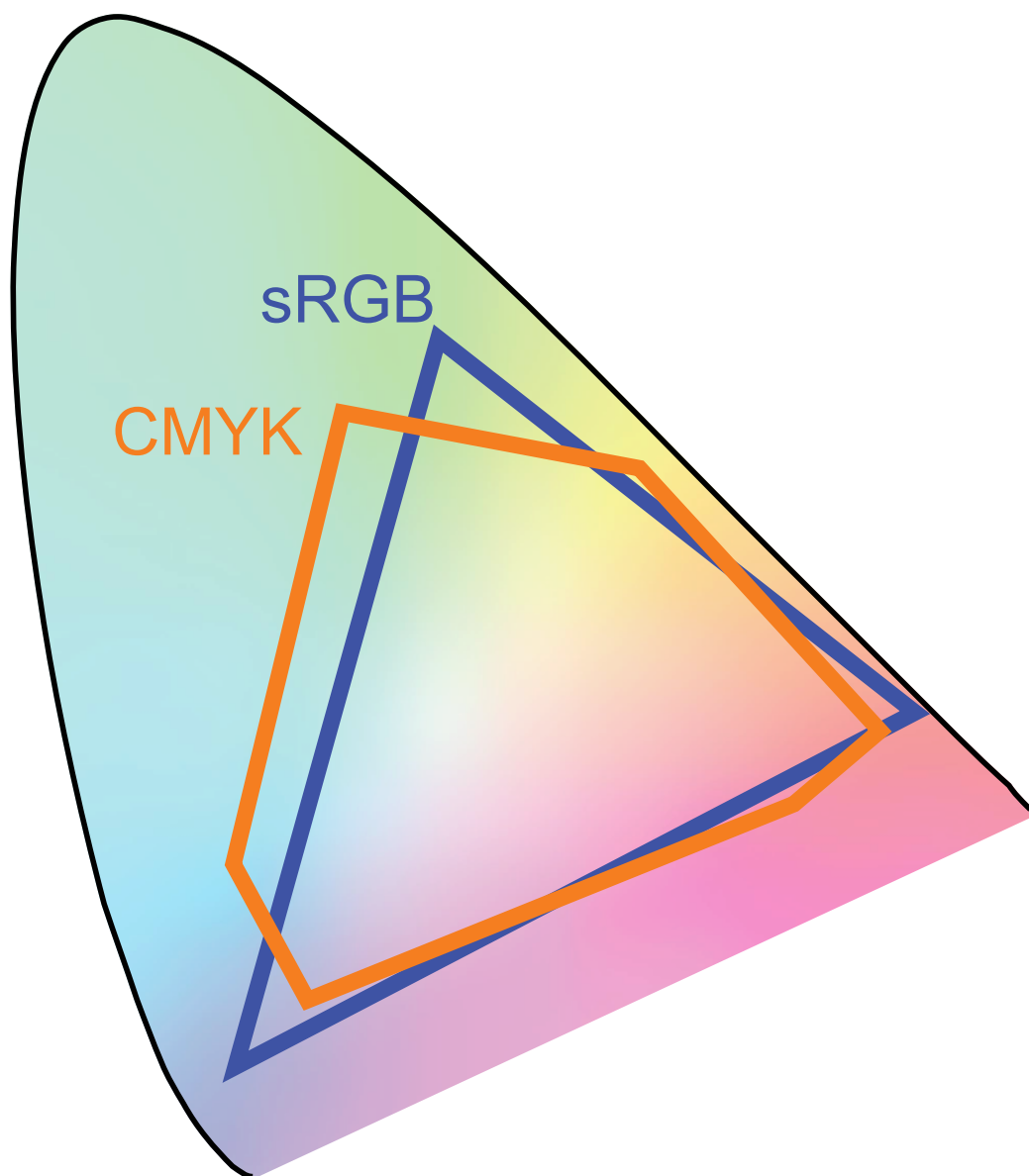
These differences between sRGB and CMYK might affect your use of colour. Every device or process for printing that you encounter will have a way of representing RGB colours using CMYK, but that is usually by shifting colours so that out-of-gamut colours can be printed (and shifting nearly out-of-gamut colours so they do not look the same as the shifted colours). If your paged media will mostly be viewed on screen, possibly with some local printing by end users, then RGB colours may be best. However, if your paged media will be commercially printed, discuss with your printer whether to use RGB or CMYK for images, etc., and who will do the final conversion to CMYK. Preparing images for commercial printing is a complex subject that is beyond the scope of this tutorial.

6.2. Text Colour : **`color`** property

Use the `color` property to specify the foreground colour of text and border colours. RGB is most commonly used for specifying colours and has three components: red, yellow, and blue. CMYK are used for print only and has four components: cyan, magenta, yellow, and black. International Color Consortium (ICC) colour profiles map between a device-independent colour space and the capabilities of a device. RGB and ICC colours are converted to CMYK when printing, although they may be retained as RGB or ICC colours in the PDF or other files produced by AH Formatter.

Colour can have the following values:

- #RGB : Specifies R, G, and B with a one-digit hexadecimal number each. (#5F0 # #55FF00)

- #RRGGBB : Specifies R, G, and B with a two-digit hexadecimal number each.

- rgb(255,0,0) : From the left, specifies R, G, and B with integer values ranging from 0 to 255.

- rgb(100%,0%,0%) : From the left, specifies R, G, and B with values ranging from 0% to 100%.

- black (and other colours) : Specifies a colour keyword. AH Formatter supports the extended colour keywords defined in CSS Color Module Level 3.

- cmyk() : Specifies a CMYK colour for four-colour process printing.

- device-cmyk() : Same as cmyk().

- rgb-icc() : Specifies a colour in a defined colour space.[10] Takes a variable number of arguments.

6.2.1. CMYK colours

CMYK colours may be specified with or without a fallback RGB colour for use with media that do not support CMYK. When the fallback RGB colour is not provided, it is calculated from the CMYK colour.

- cmyk(<C>, <M>, <Y>, <K>) : CMYK colour with Cyan, Magenta, Yellow, and Black components.

- device-cmyk(<C>, <M>, <Y>, <K>) : Equivalent defined by GCPM.

- rgb-icc(#CMYK, <C>, <M>, <Y>, <K>) : Equivalent.

- rgb-icc(<R>, <G>, <B>, #CMYK, <C>, <M>, <Y>, <K>) : CMYK colour with fallback RGB colour.

6.2.2. Opacity

RGB and CMYK colours can also be specified with an 'alpha' component specifying the opacity of the colour. It is not possible to use an 'alpha' component with a named colour.

6.2.3. `rgb-icc()`

The `rgb-icc()` colour function provides additional ways to specify colours, including:

- rgb-icc(#CMYK, 0.5, 0.5, 0.5, 0) : CMYK colour.

- rgb-icc(#Grayscale, 0.5) : Grayscale

- rgb-icc(#Separation, 'Name') : Spot colour.

- rgb-icc(#Registration) : Print with the same intensity on all separations.

- rgb-icc(#Separation, 'All') : Same as rgb-icc(#Registration).

6.2.4. Grayscale

Grayscale (monochrome) colours can be specified with rgb-icc(#Grayscale, <Scale>), optionally with extra parameters specifying a fallback RGB colour for use with devices that cannot display the grayscale colour.

6.2.5. PANTONE® spot colours

When you have the AH Formatter PANTONE® Option[11] you can specify more than 1,000 PANTONE® colours by name and have them print as a spot colour or be converted into the correct RGB or CMYK for rendering or printing.

When the formatted document is commercially printed, each PANTONE® colour can have a separate printing with the specific ink for that PANTONE® colour. The grayscale levels on the separation for each PANTONE® colour correspond to the level of tint to apply.

---

[10] `rgb-icc()` is defined in XSL 1.1 and implemented for CSS by AH Formatter.
[11] The 'AH Formatter PANTONE® Option' must be purchased separately.

Figure 16. Greyscale levels on the separation correspond to the level of tint to apply

6.2.6. Other spot colours

Spot colours can be used without the AH Formatter PANTONE® Option. However, it is necessary to provide one or both of the equivalent RGB and CMYK colours for use with media that does not support separations for spot colours. If either of the RGB or CMYK equivalent is omitted, it is calculated from the components of the other equivalent colour.

- rgb-icc(<R>, <G>, <B>, #Separation, <Name>, <Tint, <C>, <M>, <Y>, <K>) : Spot colour with name, tint, and both CMYK and RGB fallback colours.

- rgb-icc(<R>, <G>, <B>, #Separation, <Name>, <Tint) : Spot colour with name, tint, and RGB fallback colour.

- rgb-icc(<R>, <G>, <B>, #Separation, <Name>, <Tint) : Spot colour with name and RGB fallback colour.

- rgb-icc(#Separation, <Name>, <Tint, <C>, <M>, <Y>, <K>) : Spot colour with name, tint, and CMYK fallback colour.

Similarly to PANTONE® colours, when the formatted document is commercially printed, each CMYK spot colour can have a separate printing with the specific ink for that colour. The greyscale levels in the separation for each colour correspond to the level of tint to apply.

7. Counters

7.1. Numbering Chapters and Sections

Use the `counter-increment`, `counter-reset`, and `content` properties to automatically assign a series of numbers to chapter and section elements.

Use the counter name as the value of the `counter-increment` and `counter-reset` properties. Specifying a counter name for the `content` property allows the value of the counter to be inserted in the `::before` or `::after`

pseudo-elements. The counter's value increases each time an element applies `counter-increment` and resets when `counter-reset` is applied.

```
body {
  counter-reset: ChapterNo;          /* reset chapter number counter */
}
h1:before {
  counter-increment: ChapterNo;      /* add 1 to chapter number counter */
  /* Insert 'Chapter n' before chapter header  (h1)   */
  content: "Chapter" counter(ChapterNo) ":  ";
}
h1 { /* set h1:before and Chapter of h1 */
  string-set: Chapter content(before) content();
  counter-reset: SectionNo;          /* reset section number counter */
}
h2:before {
  counter-increment: SectionNo;      /* add 1 to section counter */
  content: counter(ChapterNo) "." counter(SectionNo) " ";
}
h2 { /* set h2:before and Section of h2 */
  string-set: Section content(before) content();
}
@page :left {
  @top-left { /* insert section title in the running head on the left page */
    content: string(Chapter);
  }
}
@page :right {
  @top-right { /* insert section title in the running head on the right page */
    content: string(Section);
  }
}
```

7.2. Inserting Characters : **content** property

Use `content` property to insert a string just before or after an element. With CSS 3, you can also use it to specify a string as the content of the element.

- normal : Does not insert characters.

- none : Does not insert characters. (Behaves the same as `normal`).

- string : String to be inserted is written with double or single quotes.

- url() : Specifies the URL of an image file. The content of an element can be made into an image if `content: url(image.png);` is specified.

- attr() : The specified attribute value becomes the `content` property value.

- counter() : Inserts a counter value.

- open-quote : Inserts the first pair of quotes from the `quotes` property before the element.

- close-quote : Inserts the second pair of quotes from the `quotes` property after the element.

- no-open-quote : Does not display a quotation mark but increases the level of nesting of the `quotes` property by one.

- no-close-quote : Does not display a quotation mark but decreases the level of the nesting of the `quotes` property by one.

```
.Chapter h2:before {
  content: "Chapter " counter(ChapterNo) ". ";
```

For the `content()` function that is used with the `string-set` property, see Section 4.2.2 [60]

7.3. Incrementing Counters : **`counter-increment`** property

Use the `counter-increment` property to increase the specified counter value.

- none : Does not do count.

- Counter Name : Increases the specified counter value by one.

- Counter name, space, and integers : Increases the counter value with a specified number.

```
.Chapter h2 {
  …
  counter-increment: ChapterNo;
  …
}
```

7.4. Counter Reset : **`counter-reset`** property

Use `counter-reset` property to reset the specified counter value.

- none : Does not reset count.

- Counter name : Sets the specified counter value to zero.

- Counter name, integer : Sets the counter value to the specified number.

```
.Chapter h2 {
  …
  counter-reset: SectionNo;
  …
}
```

7.5. Page counter

Use the `counter()` function to find the current page and the total number of pages.

```
<p>Number of this page = <span style="content: counter(page)"></span></p>
<p>Total number of pages in this document=<span style="content: counter(pages)">
</span></p>
```

`counter()` has an optional second argument specifying the counter style. If that is omitted, it defaults to 'decimal'.

```
<p>Number of this page =
  <span style="content: counter(page, lower-roman)"></span></p>
<p>Total number of pages in this document=
  <span style="content: counter(pages, upper-roman)"></span></p>
```

8. Lists

The `display` property does not have a value that will cause an element to display as a list. However, `display: list-item;` does cause an element to generate a list item. Every list item has a marker, which is the bullet, number, or other mark that identifies the list item. In CSS 2, the formatting of the marker is specified using the the `list-style-type`, `list-style-image`, `list-style-position`, and `list-style` properties. CSS 3 adds the `::marker` pseudo-element so that the list item marker can be styled with the full range of CSS properties and values. The `list-style-type` and `list-style-image` properties set the default contents of the `::marker` pseudo-element.

8.1. Counter styles

A "counter style" is the definition and/or implementation of the sequence of numbers, letters, and/or symbols to use to represent a numbering sequence. CSS 1 defined a handful of counter styles based on what HTML traditionally allowed on lists. CSS Counter Styles Level 3 defines the `@counter-style;` rule, which provides a mechanism for defining custom counter styles, plus it defines a number of counter styles that should all (eventually) be expected to be built into browsers.

The core of a CSS 3 counter style is that it attaches a name to an algorithm for generating string representations of integer counter values. A counter style may also include properties indicating a prefix and/or suffix to add to the generated values, additional strings to indicate negative numbers, etc. The counter style can be used in the `list-style-type` and in the `counter()` and `counters()` functions.

The following example shows a 'my-filled-circled-decimal' counter style that is a based on the 'filled-circled-decimal' counter style from CSS Counter Styles Level 3. As the name suggests, the counter style uses decimal numbers inside filled circles to represent decimal numbers. The numbers are followed by a space. The counter style is used when numbering the items in an <ol>.

```
@counter-style my-filled-circled-decimal {
system: fixed;
symbols: '\2776' '\2777' '\2778' '\2779' '\277a' '\277b' '\277c' '\277d' '\277e';
/* symbols: '#' '#' '#' '#' '#' '#' '#' '#' '#'; */
suffix: ' ';
}
ol.my-filled-circled-decimal li { list-style-type: my-filled-circled-decimal;  }
…
<ol role="my-filled-circled-decimal">
  <li title="1">1</li>
  <li title="2">2</li>
</ol>
```

8.2. Defining Custom Counter Styles : @counter-style rule

Allows definition of a custom counter style. The general form of an `@counter-style` rule is:

`@counter-style <counter-style-name> { <declaration-list> }`

Counter style names are case-sensitive, However, the names of counter styles that are predefined in CSS Counter Styles Level 3 are matched case insensitively. A counter style name cannot match "none", and "decimal" and "disc" cannot be defined as counter style names.

The following descriptors are allowed in the declaration list:

- system : Specifies which algorithm to use to construct the counter's representation.

- negative : Defines how to alter the representation when the value is negative.

- prefix : Specifies a symbol that is prepended to the marker representation.

- suffix : Specifies a suffix that is appended to the marker representation.

- range : Defines the ranges over which the counter style is defined.

- pad : Specifies a symbol with which to pad counter representations that are not a minimum number of grapheme clusters.

- fallback : Fallback counter style to be used when the current counter style cannot create a representation.

- symbols : Symbols to be used by the marker-construction algorithm.

- additive-symbols : Symbols to be used by an additive marker-construction algorithm.

- speak-as : Describes how to synthesize the spoken form of a counter.[12]

---

[12]Not implemented by AH Formatter.

8.3. Predefined Counter Styles

CSS Counter Styles Level 3 predefines some counter styles, including some that are noted as commonplace but complicated to represent with `@counter-style`. Ready-made Counter Styles, published by the W3C Internationalization Working Group, provides code snippets for user-defined counter styles for numbering systems used by various cultures around the world. For ease of reference, the Ready-made Counter Styles counter styles also duplicates the predefined styles from CSS Counter Styles Level 3.

Bibliography

So far, this tutorial has briefly introduced the practical uses of CSS page composition. AH Formatter currently implements the following related specifications:

[1] CSS 2.1 (CSS Level 2 Revision 1), W3C Candidate Recommendation, [https://www.w3.org/TR/CSS21/]

[2] CSS 3 Backgrounds and Borders, Working Draft, [https://www.w3.org/TR/css3-background/]

[3] CSS 3 GCPM (Generated Content for Paged Media), Working Draft, [https://www.w3.org/TR/css3-gcpm/]; Editor's Draft, [http://dev.w3.org/csswg/css3-gcpm/]

[4] CSS 3 Multi-column layout, Working Draft, [https://www.w3.org/TR/css3-multicol/]

[5] CSS 3 Paged Media, Final Draft, [https://www.w3.org/TR/css3-page/]

[6] CSS 3 Lists, Working Draft, [https://www.w3.org/TR/css3-lists/]

[7] CSS 3 Text, Working Draft, [https://www.w3.org/TR/css3-text/]; Editor's Draft, [http://dev.w3.org/csswg/css3-text/]

[8] CSS 3 Text Layout, Editor's Draft, [http://dev.w3.org/csswg/css3-text-layout/]

[9] CSS 3 Namespaces, W3C Candidate Recommendation, [https://www.w3.org/TR/css3-namespace/]

[10] CSS Color Module Level 3, W3C Recommendation, [https://www.w3.org/TR/css3-color/]

[11] CSS Counter Styles Level 3, W3C Working Draft, [https://www.w3.org/TR/css-counter-styles-3/]

[12] CSS Fonts Module Level 3, W3C Candidate Recommendation, [https://www.w3.org/TR/css-fonts-3/]

[13] HTML 5 — A Vocabulary and Associated APIs for HTML and XHTML, W3C Recommendation, [https://www.w3.org/TR/html5/]

[14] ISO 16684-1:2012, Graphic technology — Extensible metadata platform (XMP) specification — Part 1: Data model, serialization and core properties, ISO Standard, [https://www.iso.org/standard/57421.html]

[15] Ready-made Counter Styles, W3C Working Group Note, [http://www.w3.org/TR/predefined-counter-styles/]

[16] Requirements for Japanese Text Layout, W3C Working Group Note, [http://www.w3.org/TR/jlreq/]

[17] Requirements for Latin Text Layout and Pagination, W3C Working Draft, [https://www.w3.org/TR/dpub-latinreq/]

[18] Selectors Level 3, W3C Recommendation, [https://www.w3.org/TR/css3-selectors/]

[19] Extensible Stylesheet Language (XSL) Version 1.1, W3C Recommendation, [https://www.w3.org/TR/xsl/]

[20] Introduction to CSS for Paged Media, Antenna House, [https://www.antennahouse.com/antenna1/css/]

The Wolfenbüttel emblem2rdf Pipeline

David Maus, Herzog August Bibliothek Wolfenbüttel

Abstract

The project "Emblematica Online – Linked Open Emblem Data" set out to publish the Wolfenbüttel collection of emblem books as a resource for digital emblem studies. At the end we managed to publish information about approximately 12300 emblems and 220 emblem books.

After a short introduction into the field of digital emblem studies at the Herzog August Bibliothek Wolfenbüttel the article presents the challenges we faced when working with XML encoded information that dates back to the beginnings of web and the infancy of today's Digital Humanities, the solutions we employed to tackle these problems and how we ended up implementing a publication pipeline implemented in XProc.

Some of the topics I like to discuss are missing documentation and schema validation, legacy technology (i.e. DTD), the creative use of XML specifications, and the benefits of XProc as a declarative language for XML processing pipelines that allowed us to quickly iterate possible solutions.

1. Introduction

The Herzog August Bibliothek (HAB) is an independent research institute specializing in the study of European cultural history in the medieval and early modern period. The library's holdings from these periods form an archive of Western culture that is nearly unique in its scope. Manuscripts, incunabula, early imprints and special collections such as graphic art and maps provide material for almost limitless investigation into European cultures of knowledge in a global context.

Since 2004 the library contributes to the field of digital emblem studies in close cooperation with the University of Illinois at Urbana-Champaign and the wider digital emblem studies community. A series of projects labeled "Emblematica Online" aims to digitize, index, and do research on two of the greatest collections of emblem books worldwide. The most important outcome of the project series' efforts is the Open Emblem Portal [http://emblematica.grainger.illinois.edu/], a portal that provides access to more than 30000 emblems of six emblem book collections from the University of Illinois at Urbana-Champaign, the Herzog August Bibliothek Wolfenbüttel, Glasgow University, Utrecht University, Duke University, and the Getty Research Institute.

2. Emblematica Online – Linked Open Emblem Data

The latest iteration of the series set out to develop a core ontology for digital emblems and publish the Wolfenbüttel collection of emblems as Linked Open Data. Being an XML-centered library the goal is also to make "principled use RDF/XML" (Dodds 2016) and publish the emblems as RDF/XML documents with an accompanying RelaxNG grammar. Early on in the project we chose XProc as language both to recreate a model of the existing publication workflows and to implement the RDF/XML production pipeline.

XProc is a declarative language for XML-based processing pipelines. It was standardized by the W3C in 2010 and is currently supported by two Open Source implementations, XML Calabash [http://xmlcalabash.com] and Morgana XProc [http://xml-project.com/morganaxproc/]. XProc views a pipeline as "a sequence of operations to be performed on zero or more XML documents" (Walsh et al. 2010) and defines control structures as well as a standard library of required and optional steps that perform common operations on XML. A step being an atomic operation that, much like a pipeline, accepts zero or more XML documents as inputs and produces zero or more XML documents as output. XProc also allows for the definition of user-defined steps that combine steps of the standard library into subpipelines.

2.1. Encoding emblems

The project uses a simplified view of an emblem as an intellectual entity. An emblem is viewed as an entity with three distinct parts: A short aphorism from canonical texts of ancient writers or the Bible (*motto*), an image with meaningful, symbolic and/or allegorical elements (*pictura*), and an optional verse to be considered in conjunction with *motto* and *pictura* (*subscriptio*).

We assign a persistent identifier to every emblem using UIUC's handle system. For each emblem we provide a transcription of all language variations of the emblem's *motto*. If the emblem has a *subscriptio* we record just its language variations. E.g. we provide information that the emblem has a German or Latin *subscriptio* but not its actual content. The *pictura* is further classified with ICONCLASS [http://iconclass.nl], a specialized classification for art and iconography. With the help of ICONCLASS scholars can access emblems based on the subject represented in their *pictura*.

Two complementing XML files describe the overall structure of a digital emblem book: The `facsimile.xml` holds information about the images composing the digital facsimile. It is not a complete TEI document but a fragment with `tei:facsimile` as the outermost element and a sequence of `tei:graphic` elements. Each `tei:graphic` links to an an image of the digital facsimile with a `url` attribute. An `xml:id` attribute serves as anchor for location ladders. The `facsimile.xml` is automatically created at the end of a digitization process.

The second file, `tei-struct.xml`, encodes the general structure of the emblem book (chapters, sections, headings etc.) and marks up the emblem parts. It includes the `facsimile.xml` at the appropriate location, i.e. as sibling between the `tei:teiHeader` and the `tei:text` element, and references the pages of the digital facsimile by linking to the included `tei:graphic` elements.

An emblem part is encoded by a `tei:div` element with a `type` attribute that indicates the type of the part, a `facs` attribute pointing to a location in the emblem book, and a `n` attribute that holds the allocated emblem identifier. One or more `tei:index` elements contain the ICONCLASS classification of the *pictura*.

The structural metadata document is keyed in by undergraduates with a structural metadata editor colloquially known as TOC Editor. A structural metadata editor allows describing the structural elements of a digital facsimile, their nesting and extent, as well as page- or structure-related descriptive metadata. It is worth noting that the editor does not provide means to record the required information as such. It only provides free form text fields meant for indexing terms, so that data entry relies on a micro syntax to record the emblem identifier, the type of the emblem part, the language,

and the textual content. The editor serializes an XML-based format developed during the project Decentral Digital Incunabula Collection [http://diglib.hab.de?link=007] which in turn is transformed into the TEI encoded structural metadata document.

Figure 1. Emblem E018850 encoded in TEI

```
<div type="section" facs="#drucke_xb-4362_00111" n="58">
  <head>
  <pb facs="#drucke_xb-4362_00111" n="58"/>V. Cap.</head>
  <p>
    <pb facs="#drucke_xb-4362_00112" n="59"/>
    <pb facs="#drucke_xb-4362_00113" n="60"/>
    <pb facs="#drucke_xb-4362_00114" n="61"/>
    <pb facs="#drucke_xb-4362_00115" n="62"/>
    <pb facs="#drucke_xb-4362_00116" n="63"/>
    <pb facs="#drucke_xb-4362_00117" n="64"/>
    <pb facs="#drucke_xb-4362_00118" n="65"/>
    <pb facs="#drucke_xb-4362_00119" n="66"/>
    <pb facs="#drucke_xb-4362_00120" n="67"/>
    <pb facs="#drucke_xb-4362_00121" n="68"/>
    <pb facs="#drucke_xb-4362_00122"/>
    <index indexName="fsw" facs="#drucke_xb-4362_00122">
      <term type="structure" xml:lang="de" target="#illustration">Illustration</term>
    </index>
  </p>
  <div type="emblem_pictura" n="E018850" facs="#drucke_xb-4362_00122">
    <p>
      <index indexName="notation" facs="#drucke_xb-4362_00122">
        <term xml:lang="de" key="86(…)" type="ICONCLASS">Sprichwörter, Redewendungen, etc.</term>
        <index indexName="bsw" facs="#drucke_xb-4362_00122">
          <term xml:lang="de" key="86(…)" type="ICONCLASS">Redewendung</term>
          <term xml:lang="de" key="86(…)" type="ICONCLASS">Sprichwort</term>
        </index>
      </index>
      <index indexName="notation" facs="#drucke_xb-4362_00122">
        <term xml:lang="de" key="26C0" type="ICONCLASS">Ripa: Venti</term>
        <index indexName="bsw" facs="#drucke_xb-4362_00122">
          <term xml:lang="de" key="26C0" type="ICONCLASS">Wind</term>
          <term xml:lang="de" key="26C0" type="ICONCLASS">Eolo</term>
          <term xml:lang="de" key="26C0" type="ICONCLASS">venti</term>
        </index>
      </index>
      <index indexName="notation" facs="#drucke_xb-4362_00122">
        <term xml:lang="de" key="25G3" type="ICONCLASS">Bäume</term>
        <index indexName="bsw" facs="#drucke_xb-4362_00122">
          <term xml:lang="de" key="25G3" type="ICONCLASS">Baum</term>
        </index>
      </index>
      <index indexName="notation" facs="#drucke_xb-4362_00122">
        <term xml:lang="de" key="53A1" type="ICONCLASS">Wille; Ripa: Volontà</term>
        <index indexName="bsw" facs="#drucke_xb-4362_00122">
          <term xml:lang="de" key="53A1" type="ICONCLASS">Wille</term>
          <term xml:lang="de" key="53A1" type="ICONCLASS">volontà</term>
        </index>
      </index>
      <index indexName="notation" facs="#drucke_xb-4362_00122">
        <term xml:lang="de" key="54A7" type="ICONCLASS">…</term>
        <index indexName="bsw" facs="#drucke_xb-4362_00122">
          <term xml:lang="de" key="54A7" type="ICONCLASS">Kraft</term>
          <term xml:lang="de" key="54A7" type="ICONCLASS">Stärke</term>
          <term xml:lang="de" key="54A7" type="ICONCLASS">Macht</term>
          <term xml:lang="de" key="54A7" type="ICONCLASS">fortezza</term>
          <term xml:lang="de" key="54A7" type="ICONCLASS">forza</term>
        </index>
      </index>
    </p>
  </div>
  <div type="emblem_motto" n="E018850" facs="#drucke_xb-4362_00122">
    <p xml:lang="de">So muß es mir ergehn, Soll ich sonst fäste Stehn.</p>
  </div>
  <div type="section" facs="#drucke_xb-4362_00123">
    <p>
    <pb facs="#drucke_xb-4362_00123"/>
    </p>
  </div>
</div>
```

Structural metadata documents are then published as Emblem Schema documents to be used in the Open Emblem Portal. The Emblem Schema [http://diglib.hab.de/rules/schema/emblem] was developed to foster the exchange and aggregation of emblem book descriptions (Stäcker 2007). It closely follows Rawles' *Spine of Information headings* (Rawles 2004) and can be used both, to encode the entire structure of an emblem book (as is done in Utrecht) or to just provide a flat list of contained emblems. At the library the schema is exclusively used for that latter.

The schema defines its own vocabulary to encode emblem information but also imports elements from the Metadata Object Description Standard (MODS) [http://www.loc.gov/standards/mods/] for a bibliographic description of the emblem book, elements from the Simple Knowledge Organization System (SKOS) [http://www.w3.org/TR/skos-reference/] vocabulary for the ICONCLASS information, and elements from the TEI vocabulary as an alternative to MODS and for a structured transcription of an emblem's *motto* or *subscriptio*.

The Emblem Schema document is created in two steps. First an XSL transformation creates a list of emblems contained in a structural metadata document and adds a placeholder for the bibliographic description. A PHP script then fetches the bibliographic description from the library catalog and inserts it into the Emblem Schema document. All Emblem Schema documents are available via a web service that conforms to the Open Archive Initiative's Protocol for Metadata Harvesting (OAI-PMH) [http://www.openarchives.org/OAI/openarchivesprotocol.html].

Figure 2. The same emblem E018850 encoded as Emblem Schema

```
<emblem:emblem
  globalID="http://hdl.handle.net/10111/EmblemRegistry:E018850"
  xmlns:emblem="http://diglib.hab.de/rules/schema/emblem"
  xmlns:tei="http://www.tei-c.org/ns/1.0"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <emblem:motto>
    <emblem:transcription
      xlink:href="http://diglib.hab.de/drucke/xb-4362/start.htm?image=00122"
      xml:lang="de">
      <tei:p xml:lang="de">So muß es mir ergehn, Soll ich sonst fäste Stehn.</tei:p>
    </emblem:transcription>
  </emblem:motto>
  <emblem:pictura
    medium="engraving"
    xlink:href="http://diglib.hab.de/drucke/xb-4362/start.htm?image=00122">
    <emblem:iconclass>
      <skos:notation>86(…)</skos:notation>
      <skos:prefLabel xml:lang="de">Sprichwörter, Redewendungen, etc.</skos:prefLabel>
      <emblem:keyword xml:lang="de">Redewendung</emblem:keyword>
      <emblem:keyword xml:lang="de">Sprichwort</emblem:keyword>
    </emblem:iconclass>
    <emblem:iconclass>
      <skos:notation>26C0</skos:notation>
      <skos:prefLabel xml:lang="de">Ripa: Venti</skos:prefLabel>
      <emblem:keyword xml:lang="de">Wind</emblem:keyword>
      <emblem:keyword xml:lang="de">Eolo</emblem:keyword>
      <emblem:keyword xml:lang="de">venti</emblem:keyword>
    </emblem:iconclass>
    <emblem:iconclass>
      <skos:notation>25G3</skos:notation>
      <skos:prefLabel xml:lang="de">Bäume</skos:prefLabel>
      <emblem:keyword xml:lang="de">Baum</emblem:keyword>
    </emblem:iconclass>
    <emblem:iconclass>
      <skos:notation>53A1</skos:notation>
      <skos:prefLabel xml:lang="de">Wille; Ripa: Volontà</skos:prefLabel>
      <emblem:keyword xml:lang="de">Wille</emblem:keyword>
      <emblem:keyword xml:lang="de">volontà</emblem:keyword>
    </emblem:iconclass>
    <emblem:iconclass>
      <skos:notation>54A7</skos:notation>
      <skos:prefLabel xml:lang="de">…</skos:prefLabel>
      <emblem:keyword xml:lang="de">Kraft</emblem:keyword>
      <emblem:keyword xml:lang="de">Stärke</emblem:keyword>
      <emblem:keyword xml:lang="de">Macht</emblem:keyword>
      <emblem:keyword xml:lang="de">fortezza</emblem:keyword>
      <emblem:keyword xml:lang="de">forza</emblem:keyword>
    </emblem:iconclass>
  </emblem:pictura>
</emblem:emblem>
```

2.2. The publication process as XProc pipeline

Recreating the publication process as a sequence of XSL transformations interspersed with validation seemed straight forward. Unluckily we hit a first wall when we tried to reimplement the initial transformation with XSL. The transformation is done by a PHP script that uses a mixture of string concatenation and DOM manipulation to create the structural metadata document. From the looks of it the conversion script was developed for an earlier project (i.e. the Digital Incunabula Collection) and later modified by adding one special case after another. We decided to skip the first transformation and start with the structural metadata document instead. This didn't work either: The transformation from TEI to Emblem Schema ran but didn't return anything. As a last resort we the turned to the Emblem Schema documents and it was when some of those didn't validate against the Emblem Schema we decided to take a step back and start a deeper investigation.

The library engaged in the creation of digital content since the late nineties, specifically in the creation of resources for digital emblem studies since the early 2000s. Not surprisingly the overall biggest problem we faced was the lack of proper documentation. Although we were lucky to still have staff around that was involved in the projects back then we had to resort to a certain amount of guesswork regarding the reasons why documents looked the way they did. Human memory turned out to be unreliable and no substitute for proper documentation. Having an XML database at hand to run queries across the documents helped a lot. What struck us as odd was that even though documents were associated with a DTD or a schema the DTD or schema was not used to actually validate the documents. This meant that we had to base our guesses on existing transformation and processing scripts, the source items, and anecdotal evidence.

All in all we found the following flaws in our data and processes.

2.2.1. Data entry and initial transformation

The reliance on micro syntax to record emblem information is prone to errors. In order to e.g. state the fact that the current page shows *pictura* and *motto* of emblem E018850, and the *motto* is the German text "So muß es mir ergehn, Soll ich sonst fäste Stehn." an undergraduate has to use the code `E018850_P_M-de@So muß es mir ergehn. Soll ich sonst fäste Stehn.`.

The conversion script does not validate this micro syntax and silently drops information that doesn't fit the expected structure. We observed *motti* ending up in language tags and missing emblem parts alltogether. We assume that most of those errors are caught during data entry by a manual inspection of the emblem book's web representation, though.

More problematic are "semantic errors" that are introduced to the document structure. In some cases the conversion script adds a superfluous section after a *pictura*. Our current theory is that the section is added to hold a paragraph which in turn holds page breaks. This could be explained by a more or less informal rule for structural metadata documents that states that page break elements are only allowed inside "logical units"[1]. We are still uncertain if the addition of this superfluous section was done deliberately or is an artefact of the conversion process. We believe the latter is the case because the TEI guidelines would have provided an anonymous block element `tei:ab` without the semantics of a textual division.

Given the complicated nature of this problem and constraints of the project we decided against trying to further analyze and fix the initial conversion. We hope that replacing the ancient structural metadata editor with the one from the Kitodo software suite [http://kitodo.org] will allow us to get rid of the micro syntax and replace the PHP-based with an XSLT based transformation. What we could do is fix the most blatant errors (e.g. *motti* in language tags) and provide a list of questionable emblem structures to our expert scientist.

2.2.2. TEI to Emblem Schema

The transformation from TEI to Emblem Schema failed because the structural metadata document was declared to be a TEI document by a DOCTYPE-declaration but the TEI namespace was not declared in the document. Instead the DTD declared an `xmlns` attribute with a default value of `http://www.tei-c.org/ns/1.0`. Thus in order for the namespace binding to be present the XML processor had to process the DTD when loading the document. The reason why the transformation failed was simply that XProc used the DTD to supply the `xmlns` attribute with its default value while the original transformation was initiated in PHP which disables DTD processing by default. The transformation script thus expected the elements of the structural metadata document to be in the null namespace, while they were placed in the TEI namespace in our pipeline.

To allow for consistent namespace-aware processing we relocated the elements to the TEI namespace by applying an appropriate XSL transformation and modified the existing transformations accordingly. We also removed the

---

[1] http://diglib.hab.de/rules/documentation/structuralMD.xml: "Zu beachten ist, dass ein pagebreak pb immer innerhalb einer logischen Einheit, z.B. p, gesetzt wird und den Beginn einer Seite kennzeichnet."

DOCTYPE-declaration and changed the inclusion mechanism of `facsimile.xml` from external entity references to XInclude [http://www.w3.org/TR/xinclude/]. The latter prompted us to add a namespace declaration to `facsimile.xml`, which was missing. What we did not realize back then (and still have to do) is that we need to change references to the `tei:graphic` elements due to XInclude's base URI fixup. Referencing attributes like `facs` are typed as `xsd:anyURI` and thus need to reference the originating file and not just document local `xml:id`s.

We furthermore noticed that all structural metadata documents defined an `xml:base` attribute on the outermost element. The content of this attribute varied. Some documents used the persistent URL of the digital object, some used a relative URI reference denoting the object identifier with, and some without trailing slash. It is unclear, why `xml:base` was used in the first place and what caused the variations in its content. Effectively the attribute was used as if it holds the digital object identifier which happens to be the relative path to the object in most cases.

Although the use of `xml:base` caused no apparent error we removed it to avoid problems in the future.

2.2.3. Emblem Schema documents

Lastly we figured out why some of the Emblem Schema documents did not validate against the Emblem Schema. Unexpectedly some of the oldest documents *did* provide a transcription of an emblem's *subscriptio* as structured markup of lines and line groups. The version of the TEI vocabulary used by the then current version of the Emblem Schema did not allow line group elements (`tei:lg`) as children of a paragraph (`tei:p`). We find it interesting to note that apparently no one ever validated the documents against the Emblem Schema or found the validation error worth reporting.

We solved this problem by updating Emblem Schema to use a newer version of the TEI vocabulary that allows `tei:lg` as child of a `tei:p`.

2.3. Running the emblem2rdf pipeline

Having solved the problems with the structural metadata documents and their transformation to Emblem Schema documents we decided to base our RDF/XML on the Emblem Schema documents. This idea was driven by the plan to consolidate the structural metadata documents in the near future by removing the emblem related information and storing them as documents in their own right.

To run the final pipeline we harvest all emblem book descriptions from the OAI-PMH web service. The pipeline then iterates every emblem book description and validates it against the revised Emblem Schema. The document is skipped with a warning message if validation fails. Otherwise an XSL transformation converts every emblem from its Emblem Schema representation to RDF/XML conforming to our emblem ontology. The bibliographic description is transformed as well and added to every emblem contained in a particular emblem book. The resulting RDF/XML document is validated with a RelaxNG grammar and serialized to disk. Because we publish the emblem descriptions as static files on a web server we apply a transformation to HTML and serialize it to disk.

Due to the lack of native support for RDF serialization in XProc we use the rapper [http://librdf.org/raptor/rapper.html] command line tool to convert the resulting RDF/XML documents to NTriples, and TopBraid SHACL [https://github.com/TopQuadrant/shacl], an open source implementation of the W3C Shapes Constraint Language (SHACL) [https://www.w3.org/TR/shacl/] based on Apache Jena, to validate the RDF graph.

3. Summary

With the project "Emblematica Online – Linked Open Emblem Data" we set out to publish the Wolfenbüttel collection of digitized emblem books as Linked Open Data and utilized XProc, first to model the existing publication workflow and then to execute the transformation of TEI-encoded structural metadata documents to RDF/XML.

In the modelling phase we detect flaws in the underlying data and conversion routines. We were able to fix most of the problem and construct an XProc pipeline which turns Emblem Schema documents into RDF/XML documents. With the help of the latter pipeline we published information about approximately 12300 emblems in 220 emblem books as Linked Open Data.

Bibliography

[Dodds 2016] Dodds, Leigh 2012. Principled use of RDF/XML https://blog.ldodds.com/2012/06/12/principled-use-of-rdfxml [22.06.2016]

[Rawles 2004] Rawles, Stephen 2004. A spine of information headings for emblem-related electronic resources. In Digital Collections and the Management of Knowledge: Renaissance Emblem Literature as a Case Study for the Digitization of Rare Texts and Images. Salzburg: DigiCULT, 19–26.

[Stäcker 2007] Stäcker, Thomas 2007. Setting the Emblem Schema to Work. In Learned Love. Proceedings of the Emblem Project Utrecht Conference on Dutch Love Emblems and the Internet (November 2006). The Hague, 201–210.

[Walsh et al. 2010] Walsh, Norman, Milowski, Alex & Thompson, Henry S. 2010. XProc: An XML Pipeline Language. http://www.w3.org/tr/xproc [2016-03-19].

CREPDL: Protect Yourself from the Proliferation of Unicode Characters

Makoto Murata, Keio University and JEPA

Abstract

This paper studies machine-readable notations for describing subsets of Unicode or ISO/IEC 10646. Unicode regular expressions can describe any subset, but they have performance problems for huge subsets and cannot directly capture subsets defined in terms of other subsets. Meanwhile, the upcoming second edition of ISO/IEC 19757-7 Character Repertoire Description Language (CREPDL) overcomes these problems by providing references to well-known subsets and external CREPDL scripts.

1. Introduction

Which character in Unicode are you willing to accept? If you receive UTF-8 text from somebody, it might contain any of the 136,690 code points of Unicode 10.

Accepting any Unicode character may lead to problems in the future. First, nobody can read all characters. Second, few fonts cover all characters. Third, some software such as document editors supports only a subset of Unicode.

Historically, legacy encodings have protected users from the proliferation of characters. For example, as long as you use Shift JIS, you only have to worry about 7,000 characters. But UTF-8 now exposes almost 88,000 CJK[1] ideographic characters.

In the Unicode era, we need a language for describing which character is to be allowed and then examining text against descriptions in this language. ISO/IEC 19757-7 Character REPertoire Description Language (CREPDL) [3] [92] is an attempt of ISO/IEC JTC1/SC34 for such a language. Although the first edition was restricted to code points, the second edition can handle code point sequences, which represent grapheme clusters ("user-perceived characters").

The rest of this paper is organized as follows. In Section 2 [88] and Section 3 [88], we study subsets in Unicode and ISO/IEC 10646, respectively. In Section 4 [90], we study two existing machine-readable notations for describing subsets: regular expressions [7] [92] and a W3C notation [9] [92]. In particular, we make clear that regular expressions have performance problems for huge subsets and cannot directly capture subsets defined in terms of other subsets. In Section 5 [91], we have a quick overview of the design and implementation of CREPDL and see how it overcomes limitations of the existing notations.

2. Subsets in Unicode

The Unicode standard [5] [92] does not mandate the support of all Unicode characters. Rather, it allows implementations to support subsets of Unicode characters.

But Unicode does not define any subsets. It does not provide any mechanisms for specifying such subsets either. This is made clear by the following bullet extracted from "Interpretation" subsubclause in the subclause 3.2 (Conformance Requirements) of the Unicode standard.

> • Any means for specifying a subset of characters that a process can interpret is outside the scope of this standard.

However, it is true that Unicode regular expressions can be used for representing subsets. We will discuss this topic in Section 4.1.

3. Subsets in ISO/IEC 10646

Likewise, conformant implementations of ISO/IEC 10646 [2] [92] may support subsets rather than all characters in ISO/IEC 10646.

But ISO/IEC 10646 goes much further than Unicode. A subset is defined as either an implementation-defined list of code points, a standardized collection, or a combination of the two. Collections are standardized in Annex A of ISO/IEC 10646.

In the following subsections, we study collections in Annex A. But it is interesting to note that ISO/IEC 10646 does not provide a standard notation for specifying subsets. Different collections in Annex A are represented by different notations.

3.1. Collections

3.1.1. Code Points and Ranges

Most collections in Annex A are very simple. They are ranges of code points. For example, `LATIN-1 SUPPLEMENT` (collection 2) is a range 00A0-00FF.

`MULTILINGUAL EUROPEAN SUBSET 2` (collection 282) is more complicated. It is specified by the following ranges of code points as indicated for each row.

---

[1]Chinese, Japanese and Korean

Example 1.**MULTILINGUAL EUROPEAN SUBSET 2**

```
Plane 00
Row Values within row
00  20-7E A0-FF
01  00-7F 8F 92 B7 DE-EF FA-FF
02  18-1B 1E-1F 59 7C 92 BB-BD
    C6-C7 C9 D8-DD EE
03  74-75 7A 7E 84-8A 8C 8E-A1
    A3-CE D7 DA-E1
04  00-5F 90-C4 C7-C8 CB-CC D0-EB
    EE-F5 F8-F9
1E  02-03 0A-0B 1E-1F 40-41 56-57
    60-61 6A-6B 80-85 9B F2-F3
1F  00-15 18-1D 20-45 48-4D 50-57
    59 5B 5D 5F-7D 80-B4 B6-C4
    C6-D3 D6-DB DD-EF F2-F4 F6-FE
20  13-15 17-1E 20-22 26 30 32-33
    39-3A 3C 3E 44 4A 7F 82 A3-A4
    A7 AC AF
21  05 16 22 26 5B-5E 90-95 A8
22  00 02-03 06 08-09 0F 11-12 19-1A
    1E-1F 27-2B 48 59 60-61 64-65
    82-83 95 97
23  02 10 20-21 29-2A
25  00 02 0C 10 14 18 1C 24 2C 34 3C
    50-6C 80 84 88 8C 90-93
    A0 AC B2 BA BC C4 CA-CB D8-D9
26  3A-3C 40 42 60 63 65-66 6A-6B
    FB 01-02
FF  FD
```

Although this collection is not small, CJK collections are significantly larger. For example, `JIS2004 IDEOGRAPHICS EXTENSION` (collection 371) has 3695 code points. `BASIC JAPANESE` (collection 285) contains 6884 code points. `IICORE` (collection 370) has 9810 code points. Ranges are not useful for such CJK collections since code points in them are scattered. The definitions of these CJK collections are provided as electronic attachments.

3.1.2. Open Collections and Fixed Collections

Some collections defined in Annex A contain unassigned code points. Such collections are called *open* collections. Meanwhile, *fixed* collections contain assigned code points only.

Unassigned code points in open collections may be assigned by later versions of ISO/IEC 10646. Meanwhile, fixed collections remain unchanged in future versions.

3.1.3. References to Other Collections

Some collections are defined as the union of other collections. For example, `MODERN EUROPEAN SCRIPTS` (collection 283) is the union of more than 30 collections, each of which is a simple range. `COMMON JAPANESE` (collection 287) is defined as the union of `BASIC JAPANESE` (collection 285) and an enumerated list of 609 code points. Although it is possible to copy the definition of referenced collections and avoid references, the result would be less readable and harder to maintain.

3.1.4. Grapheme Clusters

A grapheme cluster [6] [92] is a sequence of code points that represents "user-perceived characters". A simple example is a base character followed by a combining character.

`CONTEMPORARY LITHUANIAN LETTERS` (collection 284) is the first collection containing grapheme clusters such as <004A, 0303> and <0069, 0307, 0301>. Note that 0303 is allowed to follow some code points (e.g, 004A), but is not allowed to follow others (e.g., 004B).

`MOJI-JOHO-KIBAN IDEOGRAPHS-2016` (collection 390) is a collection applicable to persons' names in Japanese public service. This collection contains grapheme clusters such as <5289,E0101> and <5351,FE00>, where E0101 is an ideographic variation selector and FE00 is a variation selector. Although E0101 is allowed to follow 5289, it is not allowed to follow other characters (5288, for example).

The size of `CONTEMPORARY LITHUANIAN LETTERS` is much smaller than that of `MOJI-JOHO-KIBAN IDEOGRAPHS-2016`. The number of code points and grapheme clusters in `CONTEMPORARY LITHUANIAN LETTERS` (collection 284) is less than 100. But the number of code points in `MOJI-JOHO-KIBAN IDEOGRAPHS-2016` is more than 52000 and that of grapheme clusters is more than 10000.

3.1.5. User-defined Subsets

3.1.5.1. Subsets Defined by Governments

Although collections in ISO/IEC 10646 are defined for technical reasons, the Japanese government defines sets of CJK ideographic characters for non-technical reasons. For example, Kyouiku Kanji [4] [92] is a set of CJK ideographic characters for elementary school education. It has 1006 characters. Another set, Jouyou Kanji, contains 2136 CJK ideographic characters for use in official government documents. Governments in Mainland China, Taiwan, Hong-Kong, and Korea also define sets of CJK ideographic characters.

3.1.5.2. Subsets for Describing Font Coverage

Characters covered by commercial fonts in Japan are slightly different from ISO/IEC 10646 collections for some reasons (backward compatibilities with Shift JIS, for example). Machine-readable descriptions of font-covered subsets make it easier to compare different fonts having different coverage.

4. Existing Machine-readable Notations for Describing Subsets

4.1. Unicode Regular Expressions

Unicode regular expressions [7] [92] can be used for representing Unicode subsets. In fact, in the Unicode Common Locale Data Repository [1] [92], subsets for each locale are represented by regular expressions. Subsubsection 5.3.3 (Unicode Sets) of Unicode Technical Standard #35 [8] [92] describes the use of regular expressions for subsets and demonstrates the use of code points, ranges, code point sequences, and set operations (union, inverse, difference, and intersection).

Any collection defined in ISO/IEC 10646 can be represented by a Unicode regular expressions. In particular, code point sequences representing grapheme clusters (e.g., `<5289,E0101>`) can be represented by regular expressions (e.g., `{\u5289\U000E0101}`).

However, there are two problems. When collections are small, these problems are insignificant. But they become quite significant for large collections such as CJK collections.

4.1.1. Referencing other subsets

Unicode regular expressions cannot reference collections defined in ISO/IEC 10646. Likewise, they cannot reference other regular expressions. Thus, regular expressions cannot directly capture collections defined in terms of other collections. It is thus necessary to create a gigantic regular expression by copying the definition of each referenced collection. This might be acceptable for `MODERN EUROPEAN SCRIPTS` (collection 283). But it is too inconvenient for `COMMON JAPANESE` (collection 287) since it references `JIS2004 IDEOGRAPHICS EXTENSION` (collection 371), which contains 3695 code points.

4.1.2. Performance

Unicode regular expression engines are slow for large collections. However, hash-based set operations are much faster. This observation is based on an experiment with ICU's Regular Expressions package[2] and a hash-based set of the programming language F#.

First, I created a regular expression for the `MULTILINGUAL EUROPEAN SUBSET 2` collection. After creating a matcher (an instance of the class RegexMatcher) from it, I invoked it for the string "　" 100000 times. In my computing environment (AMD A10-7800, 16GB Memory, Windows 10), the elapsed time was about 0.4 seconds.

---

[2] `http://www.icu-project.org/userguide/regexp`

I then created a hash-based set for the same collection and tested if it contains the same string. I did this test 100000 times. The elapsed time was about 0.015 seconds. Thus, the hash-based set is more than 20 times faster than the ICU's Regular Expressions package.

Second, I did the same experiment for the `IICORE` collection. In the case of the regular expression matcher, the elapsed time was about 23 seconds. In the case of the hash-based set, the elapsed time was about 0.015 seconds. Thus, the hash-based set is more than 1600 times faster than the ICU's Regular Expressions package.

The slow performance of regular expression engines might not be problematic if the collection is not large. But it is fatal for huge CJK collections.

4.2. A Notation for Character Collections for the WWW

"A Notation for Character Collections for the WWW" [9] [92] (hereafter W3C notation for short) provides an XML syntax for describing subsets. Although it has not become a W3C recommendation and has not been implemented, it has a number of interesting ideas.

The W3C notation does not use regular expressions. Rather, it introduces XML elements (`range` and `enum`) for representing ranges and code points, respectively.

An interesting feature of the W3C notation is its `kernel` and `hull` elements. They are used to define open collections.

Unlike regular expressions, the W3C notation is equipped with a mechanism that references other subset descriptions or well-known subsets (e.g., collections in ISO/IEC 10646). This notation can thus easily describe subsets defined in terms of other subsets.

The W3C notation also has set operations (union, inverse, difference, and intersection). They allow subsets to be defined in terms of other subsets.

However, the W3C notation lacks mechanisms for describing grapheme clusters.

5. Design and Implementation of CREPDL

5.1. Language Design

CREPDL is intended to combine the best parts of Unicode regular expressions and the W3C notation. Unlike regular expressions, CREPDL can easily handle large collections. Unlike the W3C notation, CREPDL can handle grapheme clusters.

First, CREPDL allows the use of Unicode regular expressions as atomic expressions. This is done by the `char` element of CREPDL. Note that sequences of code points, which represent grapheme clusters, can be represented by regular expressions.

Second, CREPDL borrows mechanisms of the W3C notation with some modifications.

- CREPDL allows references to collections defined in ISO/IEC 10646 and other well-known subsets. The `repertoire` element of CREPDL represents such references. For example, `IICORE` (collection 370) can be referenced by `<repertoire registry="10646" number="370"/>`.

- CREPDL allows references to other CREPDL scripts by URIs. The `ref` element of CREPDL represents such references.

- CREPDL provides set operation by the `union`, `intersection`, and `difference` elements.

- CREPDL allows open collections and fixed collections by the `kernel` and `hull` elements.

The CREPDL processor has two working modes: `character` and `graphemeCluster`. If the mode is `character`, the CREPDL processor examines each code point in the input text stream. If the mode is `graphemeCluster`, the CREPDL processor extracts grapheme clusters from the text stream by applying the algorithm as defined in [6] [92]. It then validates each grapheme cluster.

Huge well-known collections referenced by `<repertoire>` can be implemented by hash-based sets. Thus, the CREPDL processor can handle such collections very efficiently.

This paper does not cover details of the CREPDL language. Interested readers are encouraged to review the CD or upcoming DIS for ISO/IEC 19757-7.

5.2. Implementation

An open source implementation of CREPDL is available at `https://github.com/CITPCSHARE/CREPDL`. It is written in F# (a functional programming language). This implementation relies on the ICU regular expression engine.

Large collections in Annex A of ISO/IEC 10646 are implemented as hash-based sets. Validation against such collections is thus very efficient.

Another GitHub repository provides a collection of example CREPDL scripts. It is available at `https://github.com/CITPCSHARE/CREPDLScripts`.

6. Concluding Remarks and Future Works

The upcoming revision of CREPDL is intended to combine the best parts of Unicode regular expressions and the W3C notation. CREPDL is expected to work nicely for huge subsets. The F# implementation of CREPDL is publicly available.

The next step is to use CREPDL for comparing different subsets covered by different fonts. It is not easy to directly compare two CREPDL scripts. However, if there is a list of all grapheme clusters in Unicode, validation of this list against two CREPDL scripts can provide definitive answers.

Bibliography

[1] CLDR - Unicode Common Locale Data Repository http://cldr.unicode.org/

[2] *ISO/IEC 10646, Universal Coded Character Set (UCS)* .

[3] *ISO/IEC CD 19757-7:2017, Document Schema Definition Languages (DSDL) — Part 7: Character Repertoire Description Language (CREPDL).*

[4] Kyōiku kanji, Wikipedia https://en.wikipedia.org/wiki/Kyōiku_kanji

[5] The Unicode Consortium. The Unicode Standard. http://www.unicode.org/versions/latest/

[6] Unicode Standard Annex #29: Unicode Text Segmentation http://www.unicode.org/reports/tr29/

[7] Unicode Technical Standard #18: Unicode Regular Expressions http://www.unicode.org/reports/tr18/

[8] Unicode Technical Standard #35: Unicode Locale Data Markup Language (LDML) http://www.unicode.org/reports/tr35/

[9] W3C Note 14-January-2000: A Notation for Character Collections for the WWW https://www.w3.org/TR/2000/NOTE-charcol-20000114/

Rethinking transformation – the potential of code generation

Hans-Jürgen Rennau, parsQube GmbH

Hauke Brandes, parsQube GmbH

Abstract

A code generator for document to document transformation is introduced. It reduces the development effort to editing a set of metadata items attached to a tree model of the target documents. Metadata values are XQuery expressions which are typically so simple that they do not require genuine programming skills. Nevertheless, expressions are more difficult to provide than static values, and therefore possibilities of further simplifying the development task are explored, striving to enable subject matter experts to define the transformation without writing XQuery expressions. This can be achieved by generating the expressions from assertions about alignments between source and target nodes, although specific requirements will often necessitate additional information. As alignments can be represented graphically by connecting lines, the approach amounts to a solid conceptual foundation for graphical mapping tools. Finally, the underlying model of code generation driven by target document structure is generalized into a conceptual framework which is not restricted to XML data sources. Its usefulness is demonstrated by a simple code generator for transforming RDF data into XML documents.

1. Introduction

XML documents are trees of labeled information, and they may be associated with a document model (e.g. XSD), which prescribes a particular tree structure. Code for processing or constructing documents therefore often reflects the tree structure described by a document model. Such code might be called document model driven code.

While a document model implies the tree structure of its instance documents, this tree structure is usually implicit, rather than explicit, due to the use of references which "flatten" the representation [3] [112]. However, a de-normalizing transformation (replacing references by a copy of the referenced component) can generate a document model tree, a tree-structured document model which mirrors the tree structure of its instance documents as closely as possible (see [3] [112] for a detailed discussion). Augmenting the nodes of a document model tree with metadata, one obtains a document metadata tree. The metadata of a model node may convey hints how to process or create the instance document items modeled by the node. This implies a possibility to generate document model driven code from document metadata trees [3] [112]. Typically (although not necessarily) the resulting code will mirror the model tree: starting at the root node of the model and descending from each model node to its children.

An important category of document model driven code is document construction from data sources, e.g. an XML or JSON document, relational database content or RDF data. This paper explores the generation of document constructors from metadata trees in depth. Major goals are to provide a sound conceptual base and to explore its application in practice. We focus on XML document to XML document transformation, but in the last part we generalize our findings, proposing a conceptual framework for building a family of code generators, each one dealing with a different source media type.

Note on terminology. This paper leans on the concept of a document model tree as defined in [3] [112]. The term used in [3] [112] to denote a node of a model tree is location. We use the terms model node and location as synonyms.

2. Rethinking document-to-document transformation

As a starting point, we introduce a simple example transformation.

Source document:

```
resources
. books
. . @lastUpdate?
. . book*
. . . @isbn
. . . title
. . . author*
. . . py?
```

Target document:

```
publications
. @updatedAt?
. publication*
. . @publicationYear?
. . isbn
. . title
. . creator*
. . . creatorRole
. . . creatorName
```

The implementation of a transformation can use a push approach or a pull approach:

- Push approach – the code follows the data structure of the source document, letting it trigger appropriate actions
- Pull approach – the code follows the data structure of the target document, letting it trigger appropriate actions

As a rule of thumb, the push approach is more appropriate when the structure of the target document strongly depends on the actual input data, which is typically the case with document oriented XML, mainly consisting of human readable

text and structured according to its actual content. The pull approach, on the other hand, tends to be much more straightforward with data oriented XML, where the target document structure is mainly determined by the target document model, rather than unpredictable text structure.

In this discussion we focus on pull transformation, as we are interested in leveraging the information stored in the target document model. The following listing demonstrates the striking similarity between target document structure and code structure, which can be achieved using the pull approach.

Example 1. XQuery code implementing the example transformation as a pull transformation.

```
let $context := /* return

<publications>{
    let $value := $context/books/@lastUpdate return
    if (empty($value)) then () else
    attribute updatedAt {$value},
    for $context in $context/books/book return
    <publication>{
        let $v := $context/py return
        if (empty($v)) then () else
        attribute publicationYear {$v},
        <isbn>{$context/@isbn/string()}</isbn>,
        <title>{$context/title/string()}</title>,
        for $context in $context/author return
        <creator>{
            <creatorRole>{'Author'}</creatorRole>,
            <creatorName>{$context/string()}</creatorName>
        }</creator>
    }</publication>
}</publications>
```

Such code is constructed according to a simple pattern:

1. Instantiate the root element *(write start tag; content will follow)*
2. For each attribute model on this element model:
   a. Decide whether to construct this attribute
   b. If yes: set the value
3. For each child element model on this element model:
   a. Decide how many instances to construct (possibly none)
   b. For each instance:
      i. Instantiate element *(write start tag; content will follow)*
      ii. If element has simple content: set the value
      iii. If element has attributes/child elements: continue at (2)

The example illustrates that the code of a pull transformation can be regarded as a sum of two components: a scaffold which reflects the target document model and structures the code; and "nuggets" – small pieces of code pulling information from the source document. Note the frequent use of the variable $context, whose value is a shifting set of source nodes. It provides an appropriate starting point for navigating to the nodes of interest, for example:

```
$context/title
```

Here, the value is always bound to the "right" <book> element, which is the <book> currently processed. Such a context is essential for locating the appropriate source nodes.

The transformation can be regarded as a sequence of decisions, which is orchestrated by the target document structure. Main kinds of decision are:

• A?	- Whether to construct an attribute	(2a)
• #E?	- How many instances of an element to construct	(3a)
• A=?	- Which value to use for an attribute	(2b)
• E=?	- Which value to use for a simple content element	(3b.ii)

For following table compiles code snippets, stating the kind of decision they implement and identifying the source and target nodes involved.

Table 1. Code snippets taken from the example of a simple pull transformation.

Transformation code	Source nodes referenced by $context	Source nodes	Target nodes under construction	Decision kind
$context/books/@lastUpdate	<resources>	@lastUpdate	@updatedAt	A?
$context/books/@lastUpdate	<resources>	@lastUpdate	@updatedAt	A=?
$context/books/book	<resources>	<book>	<publication>	#E?
$context/py	<book>	<py>	@publicationYear	A?
$context/py	<book>	<py>	@publicationYear	A=?
$context/@isbn/string()	<book>	@isbn	<isbn>	E=?
$context/title/string()	<book>	<title>	<title>	E=?
$context/author	<book>	<author>	<creator>	#E=?
'Author'	-	-	<creatorRole>	E=?
$context/string()	<author>	<author>	<creatorName>	E=?

This example suggests a great potential for code generation based on the target document model. This is best prepared by introducing a few abstractions.

3. Source navigation based model

Now we proceed to translate the informal picture of target document construction into a model which describes document construction as composed of three primitive operations orchestrated by simple rules of iteration, invocation and value passing. The crucial operations are mappings of an input sequence of nodes to an output sequence of nodes. As nodes can be perceived as places, such mappings may be viewed as navigations – hence we call the model a source navigation based model.

3.1. Basic concepts

As discussed in the previous section, document construction can be regarded as a sequence of basic decisions concerning the questions *how many instances to construct* and *which character sequence to use* (as simple element content or attribute value). Consider, for instance, the <creator> element. As its minimum and maximum cardinality differ (being 0 and unbounded), we must take a decision how many elements we will actually instantiate. However, as there may be more than one <publication> to be constructed, the answer clearly depends on which <publication> is considered – the question "how many instances?" cannot be answered globally, but only locally, concerning <creator> children of a particular <publication> element. In fact, we must ask and answer the question again and again for each <publication> appearing in the target document. Similarly, the string value to be used for @publicationYear must be determined repeatedly, once for each <publication> element in the transformation result.

To generalize, the basic decisions concerning the instantiation of a model node are taken in the context of an individual parent node. To put it differently: if the parent node has been identified, the decisions can be made, which means one subset of the model instances can be constructed, comprising all instances found under the given parent. It is important to remember that the complete set of instances of a given model node is but the union of the subsets obtained for the instances of the parent model node. This enables a decomposition of the highly complex operation of document construction into primitive operations - small building blocks of processing which implement the basic decisions. Defining a local instantiation of a model node as the subset of its instances sharing a particular parent node, we obtain the following picture:

- Document construction = instantiation of a document model
- Document model = a set of model nodes
- Instantiation of a model node = a set of local instantiations of a model node
- Local instantiation of a model node = result of a composite operation "local construction"
- Local construction = composition of primitive operations
- Primitive operation = implementation of a basic decision

This picture amounts to a decomposition of document construction into primitive operations. However, it does not give any hints how to implement those operations. Its practical value depends on an extension enabling their implementation. In particular, if the implementation can be derived from model and configuration data, code generation becomes feasible.

Focusing on document construction by transformation of a source document, we can translate the patterns observed in the previous section into formal concepts. Table 1 showed how the basic decisions are driven by a navigation of the

source document. This navigation was described as the mapping of a context of source nodes to another set of source nodes which enables the decision. A formal definition of primitive operations is prepared by defining input, output and semantics of these mappings. Besides, having realized the importance of parent node identity, we introduce the notion of a node path as a means for expressing node identity (see Table 2 [97]).

Table 2. Some concepts facilitating the decomposition of document to document transformation into primitive operations.

Name	Abbreviation	Property of ...	Semantics
Node path	NP	Instance node	Identifies the identity of the node
Source context	SC	Node path	Given a node path, a set of source nodes enabling the construction of the target node identified by the node path
Propagated source context	$PSC_{parent-instance}$	Location	Given a location and the node path of an instance of the parent location: a set of source nodes enabling the construction of the location instances which are child of the node identified by the node path
Source context array	$SCA_{parent-instance}$	Location	Given a location and the node path of an instance of the parent location: an array with members being sets of source nodes; for each member, one location instance is constructed, with a source context equal to the value of the member

3.2. Primitive operations

Construction of the local instantiation of a model node is a composite operation which consumes as input the source context of the parent node. The operation can be implemented as a composition of three primitive operations given by the following table.

Table 3. Primitive operations serving as building blocks of the local instantiation of a model node.

Mapping name	Mapping input	Mapping output	Only applicable if ...
context-propagator	parent-instance.SC	$Location.PSC_{parent-instance}$	-
context-distributor	parent-instance.PSC	$Location.SCA_{parent-instance}$	-
context-atomizer	instance.SC	String (simple content)	L.has-simple-content=true

The context-propagator maps the source context of the parent instance to a sequence of source nodes called the *propagated context*, which is the "raw material" required for constructing the local instantiation of the current location. This raw material is used by the context-distributor to determine the number of child instances and provide each one of them with a *source context*. Each of these child instances can be constructed, as its source context is given and no other input is required: (a) if the location prescribes simple content, the text value is obtained by applying the context-atomizer to the source context; (b) if the model prescribes complex content, it is obtained by passing the source context to each attribute and child location, invoking its local instantiation.

It is interesting to note that the context-propagator is a pure navigation function: it maps an input sequence of nodes to an output sequence of nodes. Also the context-atomizer and the context-distributor consume a sequence of nodes. But while the context-atomizer produces a string, the output of the context-distributor is formally defined as an array of node sequences, with one member for each instance to be constructed. Note the special case of a location with cardinality 0..1, where the array is either empty or has a single member. In the most common case, such a context-distributor triggers an instance if the propagated context is non-empty, and it uses the propagated context as the source context of that instance.

The proposed definition of primitive operations provides a *bare minimum* of building blocks which can be composed into a complex document transformation. They should be regarded as primary building blocks which may be further decomposed into smaller units if this reduces overall complexity. For example, the mapping of the parent context to the propagated context may be defined as a sequence of three steps: an initial mapping, subsequently refined by an optional filter step, subsequently refined by an optional sorting step. Such smaller units will be defined in the context of the metadata model, as they will correspond to distinct metadata items.

3.3. Composing document-to-document transformation

The following listing provides pseudo-code of a generic function `get-local-instantiation`, which implements local instantiation: consuming a model node and the source context of a parent instance node, returning

the instances of the model node which are child node of the given parent node. Note the representation of the mapping functions as properties of the model node, using dot notation, for instance:

```
$location.context-propagator(…)
```

This reflects our intent to provide the primitive operations as metadata of the model node.

Figure 1. Pseudo code of **get-local-instantiation**, a function implementing local instantiation. An auxiliary function **new-node** is assumed to be available, which constructs a node, given the node kind, the node name and the node contents.

```
get-local-instantiation($location      as location,
                        $parent-context as node()*)
as node()* {
   $propagated-context = $location.context-propagator
                                    (parent-context = $parent-context)
   $context-array      = $location.context-distributor
                                    (propagated-context = $propagated-context)
   for $context in $context-array:
      node-for-context(location = $location,
                       context = $context)


node-for-context($location as location,
                 $context   as node()*)
as node() {
   $content =
      if $location.is-leaf:
         $location.context-atomizer(context = $context)
      else:
         for $child-location in $location.child-locations:
            get-local-instantiation(    location = $child-location,
                                        parent-context = $context)
   return
      new-node(  kind = $location.node-kind,
                 name = $location.name,
              content = $content)
}
```

Definition of a function `get-local-instantiation` amounts to a complete definition of document-to-document transformation: the transformation is achieved by applying `get-local-instantiation` to the root location of the target document. In this initial invocation, the input value of a parent context is axiomatically set to the root node of the source document.

While the primitive operations can be regarded as properties of the model nodes, the metadata expected by a code generator need not necessarily represent them one-to-one, as long as they can be *derived* from the metadata. The following section introduces a metadata model which is used by an actual code generator described later.

4. Metadata model SNAT (Source Navigation Annotated Target tree)

The source navigation based model associates each location with primitive operations, and it assumes that an implementation of these operations can be derived from metadata of the location. The next step towards building a code generator is therefore the design of an appropriate metadata model, which defines the names and semantics of metadata items. This section introduces a model which is intended to minimize the effort required for implementing a transformation. Metadata items are represented by attributes of a location tree ([3] [112]), so that a transformation is specified by editing a location tree augmented by metadata attributes.

4.1. Metadata item model

The following table describes the most important metadata attributes in terms of a name and semantics. It also indicates the scope of the attributes, which is the set of location nodes where this attribute may appear. The scope is defined by constraints imposed on the location properties. The @src item, for example, appears only on locations with simple content, and @for-each, @group-by and @sort-by appear only on locations with a maximum cardinality greater 1. By

default, all attribute values are interpreted as XQuery expressions. Exceptions are enabled by special syntax rules not described here.

Table 4. The metadata model of a SNAT (Source Navigation Annotated Target tree). Each kind of metadata is modeled in terms of an attribute name, value semantics and attribute scope, which restricts the use of this attribute to locations with certain properties. The column Gen? indicates whether the attribute is generated as part of the initial SNAT tree generated from the target XSD (Y) or added to the generated SNAT tree by hand if required (N).

Attribute name	Location properties	Meaning	Example	Gen?
alt	Location of an element or attribute which is optional	XPath expression, evaluated if @src or @ctxt yields the empty sequence; the value of @alt is used as if yielded by @src or @ctxt, respectively	"#UNKNOWN"	Y
atom	Location of a simple content element or attribute	XPath expression, evaluated in a context binding variable $v to the value of an accompanying @src; if specified, the value of @atom, rather than the concatenated string values of @src, is used as element content or attribute value	substring($v, 1, 3)	N
case	Child of a choice group descriptor (z:_choice_)	XPath expression; the selected choice branch is the first child of z:_choice_ whose @case has a true effective Boolean value	@Success eq "false"	Y
ctxt	Location of a complex element with maximum cardinality equal 1	XPath expression; its value is used as the propagated source context	PubInfo/Address	Y
dflt	Location of a simple content element or attribute which is mandatory	XPath expression, evaluated if @src yields the empty sequence; the value of @default is used as if yielded by @src	"?"	Y
for-each	Location of an element with maximum cardinality >1	XPath expression; its value is used as the propagated source context; if not accompanied by @group-by, one target instance per value item is constructed, otherwise one target instance per group of value items	books/book	Y
group-by	Location of an element with maximum cardinality > 1	XPath expression, evaluated in the context of each value item yielded by @for-each; for each group of items with equal @group-by, one target instance is instantiated	../author/surName	N
if	Location of a complex element which is optional	XPath expression, evaluated in the context of the value of an accompanying @ctxt; if specified, the element is only instantiated if the effective Boolean value of @if is true	.//(au, ed, py)	N
sort-by, sort-by2, sort-by3	Location of a complex element with maximum cardinality > 1	XPath expressions, optionally following by the string "DESCENDING", used to control the order of value items received from @for-each	author/surName	N
src	Location of a simple content element or attribute	XPath expression; selects the nodes providing the information used as element content or attribute value	author/surName	Y

An example SNAT and the transformer code generated from it are shown by the following two listings.

Example 2. SNAT document, defining the transformation described as an introductory example.

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
```

```
      <z:prolog/>
      <z:snat>
        <publications ctxt="books">
          <z:_attributes_>
            <updatedAt src="@lastUpdate" alt=""/>
          </z:_attributes_>
          <publication for-each="book">
            <z:_attributes_>
              <publicationYear src="py" alt=""/>
            </z:_attributes_>
            <isbn src="@isbn" dflt="'#MISSING'"/>
            <title src="title" dflt=""/>
            <creator for-each="author">
              <creatorRole src="'Author'" dflt=""/>
              <creatorName src="." dflt=""/>
            </creator>
          </publication>
        </publications>
      </z:snat>
</z:snats>
```

Example 3. XQuery code, generated from the SNAT tree shown in the preceding listing.

```
let $c := *
let $c := $c/books
return
<publications>{
    let $v := $c/@lastUpdate
    return
       if (empty($v)) then () else
       attribute updatedAt {$v},
    for $c in $c/book
    return
       <publication>{
          let $v := $c/py
          return
             if (empty($v)) then () else
             attribute publicationYear {$v/string()},
          <isbn>{
             let $v := $c/@isbn
             return
                if (exists($v)) then $v/string() else '#MISSING'
          }</isbn>,
          <title>{$c/title/string()}</title>,
          for $c in $c/author
          return
             <creator>{
                <creatorRole>{'Author'}</creatorRole>,
                <creatorName>{$c/string()}</creatorName>
             }</creator>
       }</publication>
}</publications>
```

The code may be viewed as assembled from the primitive operations reflecting the metadata values. All data values, for example, are obtained by evaluating the expressions found in attributes @src, @alt and @dflt. Similarly, the propagation and distribution of the source context is guided by attributes @ctxt and @for-each. The following listing summarizes the rules how to derive the implementations of primitive operations from metadata item values.

Table 5. The primitive operations context-propagator, context-distributor and context-atomizer as implied by metadata values. Notation: v@foo is the value of the expression supplied by attribute

@foo, "if v@foo" means "if the value is not the empty sequence, "if @foo" means "if attribute @foo exists". The context-distributor is described informally by equating subsets of the propagated context (given by v@for-each or v@ctxt) with the source context (SC) of a distinct target item (TI).

Context-propagator		
if @for-each:	v@for-each	
if (@ctxt and @alt):	if (v@ctxt) then v@ctxt else v@alt	
if @ctxt:	v@ctxt	
if (@src and @alt):	if (v@src) then v@src else v@alt	
else:	v@src	
Context-distributor		
if (@for-each and @group-by):	each group of items in v@for-each:	SC of one distinct TI
if @for-each:	each item in v@for-each:	SC of one distinct TI
else:	all items in v@ctxt:	SC of the only TI
Context-atomizer		
if (@atom):	v@atom	
else:	string-join($source-context, " ")	

By now we have decomposed document transformation into three primitive operations, and we have set up a model how to derive their implementation from a small set of metadata. In principle, the expressiveness of this metadata language is sufficient for describing arbitrary transformation. However, the benefits of the approach get quickly lost if the expressions supplied as metadata become very complex, or if the decomposition into independent expressions entails blunt repetition of non-trivial expressions. Therefore we extend the model by a few advanced features addressing these issues.

4.2. Advanced features

Several features extend the expressive power of the basic metadata model. These are:

- Value mappings
- User defined functions
- User-defined variables

4.2.1. Value mappings

A common requirement is the mapping of strings found in the source document to different strings to be used in the target document – for example the mapping of codes to names, or a translation between different code systems. While it would be possible to describe a value mapping by filling an @atom attribute with a switch expression, mappings are much easier to define and maintain when specified by dedicated mapping tables. Example:

```
<z:valueMap name="OperationalStatus">
  <z:entry from="true" to="Open"/>
  <z:entry from="false" to="Closed"/>
  <z:entry to="Closed"/>
</z:valueMap>
```

The document model of a SNAT document contains a prolog section where such maps can be placed. An @atom item can invoke a particular mapping use the syntax %map-$mapName, where $mapName must be replaced by the name of the appropriate value map (e.g. atom="%map-OperationalStatus").

4.2.2. User-defined functions

User-defined functions can be stored in the prolog of a SNAT document and invoked by the expressions used in the metadata attributes.

4.2.3. User-defined variables

Any location can be augmented by a binding of user-defined variables to expressions. The expression values are available when processing the descendants of the assigning node. In the following example,

```
<a:Images
```

```
        vars="group=string(. idiv 50 * 50) ; folder=concat('f', $group)"
        ...> ...
    </a:Images>
```

two variables are introduced, `group` and `folder`, whose values are available when constructing the contents of an `<a:Images>` element:

```
    <a:Section src="$folder"/>
```

4.3. Code generator SNAT

A SNAT-based code generator is available at github ([5] [112]). It generates the code of document transformations defined by a SNAT document. The SNAT is prepared by hand-editing an initial version generated from an XSD description of the target documents. After changes of the XSD, the SNAT can be upgraded automatically. The upgrade ensures that the SNAT is kept in sync with the XSD. Usually, the upgrade must be finalized by a limited amount of hand-editing, for example adding manual settings for any new items.

A typical workflow is illustrated by an example in which we assume the use of BaseX ([2] [112]) as XQuery processor. Let target documents of the desired transformation code have a root element `<publications>` and be described by an XSD `publications.xsd`. The first step is the generation of an initial version of a SNAT document (`publications.snat.xml`):

```
    basex -b "request=snat?xsd=/a/b/c/publications.xsd,
                       ename=publications"
        -o publications.snat.xml
    $HOME_XSDPLUS/xsdplus.xq
```

After editing the SNAT document, the transformation code (`publications.xq`) is generated:

```
    basex -b "request=snat2xq?snat=publications.snat.xml"
        -o publications.xq
    $HOME_XSDPLUS/xsdplus.xq
```

Given a set of input documents (e.g. `books.xml`), the transformer can be tested, creating a target document `publications.xml`:

```
    basex -i books.xml -o publications.xml publications.xq
```

After changes of the XSD, the SNAT document must be upgraded. After renaming the current SNAT document (e.g. to `publications.snat.v100.xml`), this is achieved by the following command:

```
    basex -b "request=snat?xsd=/a/b/c/publications.xsd,
                       ename=publications,
                       upgrade=publications.snat.v100.xml"
        -o publications.snat.xml
    $HOME_XSDPLUS/xsdplus.xq
```

As a new XSD version typically contains new elements and attributes, the automatic upgrade must be followed by hand-editing which supplies the necessary settings not yet made.

5. Source alignment based model

An important use case of document transformations is the consumption and construction of web service messages. Such messages often contain a variety of information items with subtle semantics, best understood by subject matter experts. These experts play a key role in the correct implementation of transformations. How can their contribution be made as efficient as possible?

Creating a SNAT tree does usually not require genuine coding skills. Most entries are simple path expressions identifying source items by an item name or a path which is a sequence of item names. Nevertheless, as a rule also a few more complex expressions are required, which would be difficult to supply for a person who is not a software developer.

The question arises if SNAT trees might themselves be *generated* from simpler input, which can be provided by subject matter experts without an IT background. The core of their expertise is a thorough understanding of item semantics on

both sides, source and target. The expert recognizes the alignments between source items and target items, and in this section we explore possibilities of leveraging this competence in an optimized way. The idea is to obtain SNAT documents from an identification of alignments and a minimum of additional information.

5.1. What is an alignment?

We use the term *alignment* to mean a directed relationship of dependency: a target location is aligned with a source location when the construction of target nodes which are target location instances requires information about the presence or content of source nodes which are source location instances. There may be a semantic equivalence between the aligned locations, but this is not necessarily the case. In our introductory example of a transformation, the target location `creator` which is child of a `publication` location may be regarded as semantically equivalent to an `author` location found under a `book` location. As a counter example, consider a target location `airline` whose nodes are constructed by extracting information from flight numbers provided by `flightNumber` elements: the target location `airline` depends on the source location `flightNumber`, but there is obviously no semantic equivalence.

5.2. Representation versus information

Alignments can be represented graphically: by lines connecting the symbol of a target location with the symbols of one or more source locations. This possibility enables graphical mapping tools as offered by various products (e.g. [1] [112]). But while graphical representation is very helpful for human understanding and can facilitate human decisions, it is only a representation and cannot be equated with the information content provided by the alignments. Technically speaking, it is a frontend used for collecting information, to be distinguished from the information itself.

5.3. Annotated target tree (SAAT)

We propose to model alignments as metadata describing target locations. The alignments of a given target location consist of source location identifiers. A readible form of these are name paths like `/resources/books/book`. A straightforward representation of the alignments describing a document-to-document transformation is a location tree of the target documents, augmented by additional attributes on the location nodes which convey the name paths of the aligned source locations. Such a document we call a SAAT – Source Alignment Annotated Target tree. Here comes an example, in which the attributes supplying alignments are named `al`:

```
<z:saats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:saat>
    <publications al="/resources/books">
      <z:_attributes_>
        <updatedAt al="/resources/books/@lastUpdate"/>
      </z:_attributes_>
      <publication al="/resources/books/book">
        <z:_attributes_>
          <publicationYear al="/resources/books/book/py" />
        </z:_attributes_>
        <isbn al="/resources/books/book/isbn"/>
        <title al="/resources/books/book/title"/>
        <creator al="/resources/books/book/author">
          <creatorRole/>
          <creatorName al="/resources/books/book/author"/>
        </creator>
      </publication>
    </publications>
  </saat>
<saats>
```

Compare this to the SNAT tree specifying the transformation precisely, in which all attributes describing navigation are highlighted:

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:snat>
    <publications ctxt="books">
      <z:_attributes_>
```

```
      <updatedAt src="@lastUpdate" alt=""/>
    </z:_attributes_>
    <publication for-each="book">
      <z:_attributes_>
        <publicationYear src="py" alt=""/>
      </z:_attributes_>
      <isbn src="@isbn" dflt="'#MISSING'"/>
      <title src="title" dflt=""/>
      <creator for-each="author">
        <creatorRole src="=Author" dflt=""/>
        <creatorName src="." dflt="'?'"/>
      </creator>
    </publication>
  </publications>
  </snat>
<snats>
```

In a SNAT tree, navigation is described by expressions, to be evaluated at runtime in a current context implied by the preceding steps of navigation. Alignments, on the other hand, are static identifiers, not expressions (in spite of the appearance). Nevertheless, the example exhibits a striking correspondence between the name paths describing alignments and navigation expressions.

Table 6. A comparison between navigation steps and alignments.

Target location	Alignment	Navigation	Navigation attribute
`publications`	/resources/books	/resources/books	ctxt
`. @updatedAt`	/resources/books/@lastUpdate	@lastUpdate	src
`. publication`	/resources/books/book	book	for-each
`. . @publicationYear`	/resources/books/book/py	py	src
`. . isbn`	/resources/books/book/isbn	isbn	src
`. . title`	/resources/books/book/title	title	src
`. . creator`	/resources/books/book/author	author	for-each
`. . . creatorRole`	-	-	-
`. . . creatorName`	/resources/books/book/author	.	src

Corresponding values of alignment and navigation are marked by a subset relationship: the navigation leads to a set of nodes which is a subset of the nodes represented by the corresponding alignment. So, for example, navigation `author` (which is short for `child::author`) is evaluated in the context of an individual `book` element and selects all `author` nodes found under that `book` node. This is a subset of the nodes represented by the alignment `/resources/books/book/author`, which is the set of *all* `author` nodes found under any `book` node.

5.4. Mapping alignments to navigation

A SNAT document describes an intended transformation precisely, whereas alignment is less precise. Consider, for example, the relationship between `publication`, `isbn` and `title` nodes. The SNAT document guarantees that each source `book` is mapped to a distinct `publication` and that the `isbn` and `title` children of a `publication` element refer to the same `book` source element. This expectation is very intuitive, and therefore it may go unnoticed that the alignments shown above do not make these commitments: the alignments would not be contradicted by a transformation where a `publication` contains a `title` taken from one `book` and an `isbn` taken from another `book`, or (if we ignore cardinality constraints) a `publication` containing all `title` and all `isbn` values found in any book.

The core of this problem lies in semantic relationships pertaining to structural relationships in general, and parent-child relationships in particular. The node name `title`, for instance, indicates that node contents represent a title, but it does not hint at a semantic relationship between nodes, which constrains the child `title` to represent a property of the parent `publication`. (Were the element name `hasTitle`, this would be slightly different.) The SNAT model of a transformation preserves the intended semantic relationships implicitly: via navigation. Navigation to a *particular* `title` as well as to a particular `isbn`, executed while constructing an individual target `publication`, ensures that the target document expresses the intended semantics.

The question arises if we cannot set up rules how to map alignments to navigation steps. Table 6 [104] clearly suggests such a possibility. It should be remembered that the SNAT model is essentially a decomposition of transformation into navigation steps, or, the other way around, an integration of navigation steps into a complete picture of a transformation. If we can infer navigation steps from pairs of alignments (the alignments of parent and child locations), we have come very close to solving the problem of mapping alignments to a complete picture of the transformation. The task of generating transformation code from alignments can be redefined as the task of mapping a SAAT tree to a SNAT tree.

5.5. Inferring context propagation

As discussed in previous sections, the SNAT model of a transformation defines three primitive operations – context-propagator, context-distributor, context-atomizer. Among these, it is the context-propagator which represents navigation: the mapping of a set of source nodes to another set of source nodes. The input of this mapping is the source context of an individual target node; the output of this mapping is the (collected) source context of the child nodes of this target node.

5.6. Semantic versus structural relationships

Consider the construction of a `publication` node, which has a single `book` node as source context. The context-propagator of the `title` child location should *not* select all `title` nodes found in the source document, but a *particular* `title` – the one which is child of the particular `book` providing the source context of the `publication`. The requirement captures a semantic relationship between the `publication` and `title` nodes under construction. If target node P and target node C have a certain semantic relationship R (here: C gives the title of P), then the source context of P and the source context of C should have the same semantic relationship R (the source context of C should give the title of the source context of P). Mapping alignments to navigations (thus: SAAT to SNAT) means selecting those instances of an alignment which have a particular semantic relationship with the source context of the parent instance. The semantic relationship is implicit - implied by a structural relationship, the parent-child relationship defined as part of the target document model.

As the semantic relationship guiding navigation is implicit, implied by a structural relationship between target nodes (parent-child), it must also be derived from a *structural relationship* between sources nodes. So we have arrived at the general question: how is the semantic relationship pertaining to a particular parent-child pair in the target document model translated into a structural relationship connecting source nodes, more precisely: leading from the source context of the parent node to a subset of the source alignment of the child location?

We start by considering a particular case for which an intuitive solution is easily found. Let P and C denote two target locations, where C represents child nodes of P. (Example: P = `publication`, C = `title`). Assume that the alignment of C is a child location of the alignment of P. (Example: alignment of P: `book`, alignment of C: `title`.) Then we may expect the parent-child relationship defined by the target document model to have similar semantics as the parent-child relationship between their alignments, as defined by the source document model. (Example: the child gives the title of the parent.) So we set up a first, preliminary rule: "if the alignments of parent and child locations are themselves parent and child locations, then set the context-propagator of the target child location to select all instances of the source child location which are child nodes of the source context of the parent node." In our example, this rule would yield the desired result: the source context of the target `title` would be the child node of the source node serving as the source context of the target `publication` node.

Parent-child relationships are often modified by the insertion of intermediate nodes providing some sort of grouping. Therefore we might generalize our preliminary rule to select all instances of the target child locations's alignment which are *descendant* of the target parent node's souce context. While this will cover many cases, this is a far cry from a general solution, as hierarchical relationships need not be preserved by the transformation. As an example, consider target documents focusing on authors and describing publications as child elements of the node representing an author

5.7. Shortest-path-principle

A general approach to the mapping of alignments to navigations must be based on a generalized way of mapping semantic relationships assumed in the target model to structural relationships observed in the source model. Given a parent and a child location from the target model, we assume a semantic relationship between these locations and we also assume a similar semantic relationship to exist between the alignments of the target locations - between the alignment of the target parent location and the alignment of the target child location. The important point is that the cases where the alignments of a parent-child pair is itself a parent-child pair or an ancestor-descendant pair are handled as expected, yet regarded as special cases of a general rule. The approach requires a precise definition of the structural relationship existing between two arbitrary sets of nodes, supplied by the alignments of target parent and target child location. The definition is expressed as a navigation mapping a given node from the target parent alignment to a set of nodes from the target

child alignment. The definition of the rule is ultimately arbitrary, and the challenge is to find a solution which matches intuitive expectations best and under the broadest set of circumstances.

We define the structural relationship between the alignments of parent and source locations as the shortest navigation path leading from the target parent alignment (which is one or more source locations) to the target child alignment (one or more source locations). The shortest path between two nodes A and B is defined as follows: (a) if A is descendant (ancestor) or B, the shortest path is along the descendant:: (ancestor::) axis; (b) otherwise the shortest path is a prefix leading from A to the nearest common ancestor C, followed by the path from C to B along the descendant:: axis. The following table compiles a few examples.

Table 7. Application of the "shortest-path-principle".

Source alignment of parent node	Source alignment of child node	Shortest path
A/B/C	A/B/C/D	D
A/B/C	A/B/C/D/E	D/E
A/B/C	A/B	parent::B
A/B/C	A	parent::B/parent::A
A/B/C	A/B/X/Y	parent::B/X/Y

5.8. Future work

The shortest-path-principle needs elaboration. When the alignments of the target parent node comprise several source locations, inferred navigation is not the union of all navigations from any parent alignment to any source alignment. To illustrate the problem, let us assume that the alignments of `publication` comprise `book` and `journal`, and the alignment of child location `title` comprise `title` under `book` and `journalTitle` under `journal`. Given a parent `publication` with a source context consisting of a `book` node, we want to rule out the shortest path from `book` to `journalTitle`. The example shows that the shortest-path-principle requires further elaboration which is work in progress.

6. Metadata model SAAT (Source Alignment Annotated Target tree)

In the preceding section we have shown that alignments can be mapped to navigation which plays the role of context propagation, as defined by the SNAT model of document transformation. What is the potential value of a location tree augmented by metadata providing source alignments? This would be the minimal version of a SAAT tree, a Source Alignment Annotated Target tree.

6.1. Minimal SAAT

A SAAT tree can be mapped to a SNAT tree, and therefore it implies a precisely defined document to document transformation:

- *Propagation of the source context* is derived from source alignments according to the shortest-path-principle; this enables populating the SNAT metadata related to context propagation (@ctxt, @for-each, @src)

- *Distribution* of the propagated source context over target instances is decided by a default rule (one target instance per instance of the source context, unless the maximum cardinality of the target is one); SNAT metadata @group-by and @sort-by are not used

- *Atomization* of the source context is decided by a default rule (string values of all context nodes are concatenated, using a blank as separator)

No matter how sensible the defaults are, they are only defaults, meeting the actual requirements in some cases and not meeting them in others. A minimal SAAT does define a transformation precisely, but it cannot be used for expressing arbitrary transformations.

6.2. Alignment qualifiers

This limitation may be addressed by alignment qualifiers – metadata modifying the interpretation of the alignments. In particular, the propagation, distribution and atomization of the source context might be adapted to special requirements not covered by the default rules. This approach requires a metadata model defining the names and semantics of "aqua items", metadata items playing the role of alignment qualifiers. Embarking on this task, one should remember a basic

difference between the SNAT model and the SAAT model: the SNAT model is based on expressions and closer to a software developer's perspective; the SAAT model focuses on alignments which can be identified by a subject matter expert. Simplicity is an important benefit of the SAAT approach, without which it might not be worthwhile, considering the superior expressiveness of the SNAT model. The definition of a SAAT-based metadata model may be perceived as the challenge to find a tradeoff between simplicity and expressiveness.

An important usecase of SAAT models is the support of graphical mapping tools: the graphical representation of a document to document transformation can be captured by a SAAT tree, which can in turn be mapped to a SNAT tree providing both – a specification of the mapping semantics and an implementation of mapping source code.

The construction of a SAAT based metadata model is work in progress, geared towards supporting a graphical mapping tool which is part of a framework supporting data transformations between large type systems.

6.3. Code generator SAAT

A simple SAAT-based source code generator is integrated into the SNAT-based source code generator described in Section 4.3 [102]. Currently, the metadata model is restricted to alignments (@al attributes), and alignment qualifiers are not yet supported. Behind the scenes, code generation is based on a transformation of the supplied SAAT document into a SNAT document. Let target documents of the desired transformation code have a root element `<publications>` and be described by an XSD `publications.xsd`. An initial version of a SAAT document is generated by the following call:

```
basex –b "request=saat?xsd=/a/b/c/publications.xsd,
                    ename=publications"
      –o publications.saat.xml
$HOME_XSDPLUS/xsdplus.xq
```

After editing the SAAT document, the transformation code (`publications.xq`) is generated:

```
basex –b "request=saat2xq?saat=publications.saat.xml"
      –o publications.xq
$HOME_XSDPLUS/xsdplus.xq
```

Given a set of input documents (e.g. `books.xml`), the transformer can be tested, creating a target document `publications.xml`:

```
basex –i books.xml –o publications.xml publications.xq
```

After changes of the XSD, the SAAT document must be upgraded. After renaming the current SAAT document (e.g. to `publications.saat.v100.xml`), this is achieved by the following command:

```
basex –b "request=saat?xsd=/a/b/c/publications.xsd,
                    ename=publications,
                    upgrade=publications.saat.v100.xml"
      –o publications.saat.xml
$HOME_XSDPLUS/xsdplus.xq
```

Like an upgraded SNAT document, an upgraded SAAT document must usually be hand-edited in order to supply the settings required for new element and attributes.

7. AT Map Machine

The previous sections explained the use of annotated target tree models for generating the source code of document transformations. Now we explore the possibility of generalization, enabling the transformation of non-XML data sources into XML documents. We propose the idea of an annotated target tree as a concept supporting the generation of various kinds of data mappers.

7.1. Does SNAT presuppose XML?

A SNAT-based code generator is driven by metadata enabling primitive operations, which serve as the functional building blocks of a document transformation. These operations map a node sequence to a value which is another node sequence (context-propagator), an array of node sequences (context-distributor) or a string (context-atomizer). Does the central role played by node sequences mean that the SNAT model is restricted to XML data sources?

The apparent limitation can be overcome if we generalize the notion of a "node sequence" to mean a *sequence of distinct items*. The following table gives a few examples.

Table 8. Examples for a possible generalization of the node concept, meaning distinct items of which the resource is composed.

Source data media type	What is a "node"?
HTML	DOM node
JSON	JSON item (object, array, string, number, boolean)
CSV	table, row, cell
SQL	db, table, row, column
RDF	RDF node

For each media type based on distinct items of information a SNAT-based code generator may be defined, following a scheme of actions discussed below. Note, however, that with some media types (e.g. HTML, JSON, CSV) a simpler approach to transformation is to use an XML representation of the original input, readily obtained by Open Source products (like BaseX ([2] [112])) with its extension functions for parsing non-XML resources into XML). Such a trivial preprocessing step enables the reuse of code generators for XML-to-XML transformation (like the one described in Section 4.3 [102]) for non-XML data sources.

7.2. Does SAAT presuppose XML?

A source alignment based model (as defined in section "Source alignment based model [102]") is a set of rules how to generate a SNAT document from a SAAT document. A SAAT document describes the alignment of target locations with source locations and thus presupposes a location tree describing the source data set. However, the concept of a location tree has not yet been defined for non-XML resources. For this reason, an exploration of SAAT-based code generation for the transformation of non-XML data sources is beyond the scope of this paper.

7.3. Scheme for building SNAT-based code generators

The concept of a SNAT-based code generator spans two layers:

- The ground layer is the decomposition of document construction into primitive operations
- The top layer is a metadata model enabling the implementation of those primitives

The ground layer is independent of a particular source data media type, as long as the source data can be viewed as a collection of distinct items. The metadata model, on the other hand, must specify the construction of code which can be applied to sets of source data items. Such code will usually rely on a particular media type. We conclude that

- A SNAT-based code generator is tied to a particular source data media type
- The dependency is restricted to the metadata model

Note. As mentioned before, the dependency on a single media type can be removed by a preliminary step translating various media types into a single media type whose transformation is natively supported. For example, an XML-dependent code generator can be easily extended to support JSON, CSV and HTML data sources.

In general, the task of designing a SNAT-based code generator can be redefined as the design of a metadata model which enables a generic implementation of the primitive operations using metadata as sole input. The model comprises three submodels:

- metadata item model – defines the names and semantics of metadata items
- metadata value model – defines the interpretation of metadata item values
- code assembly model – defines how to assemble the source code of primitives from metadata

7.3.1. Metadata item model

Metadata items fall into two categories: *location* metadata items and *prolog* metadata items. Location metadata items are properties of an individual location. Prolog metadata items set up a context facilitating the specification and interpretation of location metadata items. The metadata model described in section "Metadata model SNAT [98]", for instance, defines various kinds of location metadata items (see table "Metadata items SNAT [99]") and two kinds of prolog metadata.

7.3.2. Metadata value model

The metadata model must specify how to interpret the metadata item values. Note that the interpretation may depend on the kind of metadata item. There are many possibilities, for example:

- Expressions from a standardized query language (XQuery, SQL, SPARQL, …)
- Code from an imperative programming language (Java, Python, …)
- Expressions from a purpose-defined language
- Data structures (XML, JSON, CSV, …)
- Atomic values (strings, numbers, …)
- Invocations passing arguments to functions defined by the metadata model
- Any combination of the above

The metadata value model described in section "Metadata model SNAT [98]" uses the following combination:

- Location metadata items: the value is interpreted as XQuery expression, unless syntactically marked up as invocation of a mapping function (reflecting a user-supplied value mapping) or one of a set of functions defined by the metadata model (functions not described in this paper)
- Prolog metadata items: the value is interpreted as a data structure (item kind `valueMap`) or an XQuery function (item kind `function`)

7.3.3. Code assembly model

The code assembly model is a set of rules which map location metadata items to implementations of the three primitive operations defined by the SNAT model - context-propagator, context-distributor and context-atomizer. As an example, table "The assembly of primitive operations [100]" summarizes the code assembly model of the code generator described in section "Metadata model SNAT [98]".

7.4. Using SNAT as a meta model

We have introduced the concept of a SNAT (Source Navigation Annotated Target tree) as a document which specifies the transformation of source data sets of a particular kind into instances of a particular document model. A SNAT document is a location tree obtained from the target document model and augmented by metadata. The names, semantics and scope of the metadata items are defined by a metadata model (see for example table "Metadata items SNAT [99]"). Relying on such a metadata model, a SNAT-based code generator transforms a SNAT document into executable code implementing primitive operations and composing them into the transformation as a whole.

Defined in such general terms, the concept of a SNAT-based code generator becomes an abstract model which may guide the design of concrete code generators, distinguished by a particular metadata model and expecting source data with a particular media type. To illustrate this possibility, we introduce a SNAT-based code generator for constructing XML documents from RDF data.

8. Proof of concept: RDF-to-XML, SNAT-based

This sections sketches a simple code generator for RDF-to-XML transformations, which is driven by a Source Navigation Annotated Target tree. As a SNAT-based code generator is distinguished by a particular metadata model, the description can be structured according to the main parts of the metadata model.

8.1. Metadata item model

Although dealing with RDF data sources, we may reuse the metadata item model introduced in section "Metadata model SNAT [98]" virtually unchanged. Again, table "Metadata items SNAT [99]" gives us the names of the most important metadata items and describes the role of the item values during document construction. For example, @ctxt and @for-each items provide the mapping of a current context (set of RDF resources) to a new context (another set of RDF resources); and @src items map a context to the RDF nodes supplying data values. Note, however, that with RDF data sources the metadata item *values* cannot be XQuery expressions, as RDF nodes are not part of the XQuery data model, so that XQuery expressions cannot consume RDF nodes.

8.2. Metadata value model

The new code generator must therefore define a *new metadata value model*. A possible approach would be to define metadata item values to be SPARQL expressions. This would work, as SPARQL expressions are capable of mapping

a current context of RDF nodes to the output required by context-propagator, -distributor and –atomizer. But the difference between XQuery and SPARQL expressions with regard to simplicity and intuitiveness should be considered. With XQuery, typical annotations are small path expressions, which are easy to write and to understand. Dealing with SPARQL equivalents, this ease of writing and reading will be lost, and it must be questioned if a tree studded with SPARQL expressions will be appreciated as a straightforward way of describing an RDF to XML transformation.

Guided by these considerations, we designed a simple path notation for specifying navigation within an RDF graph in a way which is similar to XPath navigation. This tiny expression language, RPath, reuses the basic concepts of XPath: navigation axes, name tests and predicates. The following listing gives a few examples of RPath expressions and their translation into SPARQL queries.

Example 4. Examples of SPARQL queries generated for RPath expressions.

```
###
### SPARQL generated for RPATH:
###     lib:author
###
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context lib:author ?newContext .
FILTER (?context IN ( a:n1.1.1, a:n1.1.2 )) . # the current context

###
### SPARQL generated for RPATH:
###     //lib:book/lib:author
###
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX zzz: <http://www.xsdplus.org/ns/internal#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context (!(zzz:NEVER)+)/lib:book ?value1 .
?value1 lib:author ?newContext .
FILTER (?context IN ( a:root-elem )) . # the current context

###
### SPARQL generated for RPATH:
###     ancestor::lib:books
###
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX zzz: <http://www.xsdplus.org/ns/internal#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context ^(!(zzz:NEVER)+) ?newContext .
?newContext ^!(zzz:NEVER)/lib:books ?newContext .
FILTER (?context IN ( a:n1.1.1, a:n1.1.2 )) . # the current context
```

Using RPath one may concisely express graph navigation as it is typical when mapping RDF data to XML. However, RPath does not support advanced requirements. When they surface, SPARQL queries can be used as fallback: the metadata value model allows the use of both, RPath expressions and SPARQL queries.

8.3. Code assembly model

The code assembly model uses XQuery as the target language, and the generated code is very similar to the code generated for the transformation of XML input. The fact that RDF nodes, rather than XML nodes, play the role of key intermediates (the source context) does not pose a problem. RDF resources are represented by their URI (a string), and RDF values are represented by their string representation. The code generator translates any RPath expressions into SPARQL queries, collects all SPARQL queries in a library and associates each one with a query key. The generated XQuery code submits the appropriate query key along with the current context to a generic function (updContext) which resolves the query to a new context or a single data value, dependent on the primitive operation being executed.

8.4. Example SNAT and source code

A small example will illustrate the use of the code generator. Let the goal again be the construction of a `publications` document, but this time using an RDF data source. The following two listings show a SNAT document and the source code generated from it.

Example 5. SNAT document, defining the transformation of RDF data into an XML document.

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:prolog>
    <z:nsMap>
      <z:ns prefix="lib" uri="http://www.example.org/ns/resources/library#"/>
    </z:nsMap>
  </z:prolog>
  <z:snat>
    <publications ctxt="$libs"
      <z:_attributes_>
        <updatedAt src="lib:lastUpdate" alt=""/>
      </z:_attributes_>
      <publication for-each="descendant::lib:book">
        <z:_attributes_>
          <publicationYear src="lib:py" alt=""/>
        </z:_attributes_>
        <isbn src="lib:isbn" dflt="'#MISSING'"/>
        <title src="lib:title" dflt=""/>
        <creator for-each="lib:author">
          <creatorRole src="'Author'" dflt=""/>
          <creatorName src="." dflt=""/>
        </creator>
      </publication>
    </publications>
  </z:snat>
</z:snats>
```

Example 6. XQuery code, generated from the SNAT document shown in the preceding listing.

```
declare variable $dataSources as element(rx:dataSources) external;
declare variable $sparqlLib :=
<sparqlLib>
  <sparql key="$libs" initialContext="$libs"/>
  <sparql key="descendant::lib:book">...</sparql>
  <sparql key="lib:author">...</sparql>
  <sparql key="lib:books/lib:lastUpdate">...</sparql>
  <sparql key="lib:isbn">...</sparql>
  <sparql key="lib:py">...</sparql>
  <sparql key="lib:title">...</sparql>
<sparqlLib>;

declare function f:updContext($dataSources as element(rx:dataSources),
                              $context as xs:string*,
                              $queryKey as xs:string) as xs:string* {...};

let $c := f:updContext($dataSources, (), "$libs")
return
<publications>{
    let $v := f:updContext($dataSources, $c, "lib:books/lib:lastUpdate")
    return
        if (empty($v)) then () else
        attribute <updatedAt {$v},

    for $c in f:updContext($dataSources, $c, "descendant::lib:book")
```

```
        return
            <publication>{
                let $v := f:updContext($dataSources, $c, "lib:py")
                return
                    if (empty($v)) then () else
                    attribute publicationYear {$v},
                <isbn>{
                    let $v := f:updContext($dataSources, $c, "lib:isbn")
                    return
                        if (exists($v)) then $v else '#MISSING'
                }</isbn>,
                <title>{f:updContext($dataSources, $c, "lib:title")}</title>,

                for $c in f:updContext($dataSources, $c, "lib:author")
                return
                    <creator>{
                        <creatorRole>Author</creatorRole>,
                        <creatorName>{$c}</creatorName>
                    }</creator>
            }</publication>
}</publications>
```

The XQuery source code has the same structure as the code generated for XML data sources (compare example "XQuery code generated for XML data source [95]". Note the difference - using an XML data source, navigation is accomplished by path expressions, whereas using an RDF data source, navigation is accomplished by submitting SPARQL queries referenced by query keys to a generic evaluator. The original RPath expressions are used as query keys, which enhances the readability of the code.

The input of the transformation is an XML document identifying the RDF graph and a set of resources bound to user-defined names. Example:

```
<rx:dataSources xmlns:rx="https://www.xsdplus.org/ns/...">
  <rx:namespace prefix="lib" uri="http://www.example.org/ns/..."/>
  <rx:graph name="main" uri="localhost:9093//...">
    <rx:resource uri="lib:rwth" name="libs"/>
    <rx:resource uri="lib:wage" name="libs"/>
    <rx:resource uri="lib:jota" name="libs"/>
  </rx:graph>
</rx:dataSources>
```

The metadata expressions can reference the resources via variable references (e.g. `ctxt="$libs"` and thus use them as starting points of navigation.

Bibliography

[1] *Altova MapForce - tool for data mapping, conversion and ETL.*. Altova. 2018. https://www.altova.com/de/mapforce.

[2] *BaseX – The XML Framework. Lightweight and High-Performance Data Processing.*. BaseX. 2018. http://basex.org.

[3] *Location trees enable XSD based tool development.*. Hans-Juergen Rennau. 2017. http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf.

[4] *SPARQL 1.1 Query Language.* World Wide Web Consortium (W3C). 2013. https://www.w3.org/TR/sparql11-query/.

[5] *xsdplus - a toolkit for XSD based tool development.* Hans-Juergen Rennau. 2017. https://github.com/hrennau/xsdplus.

Non-XML workflows with XProc 3.0

Achim Berndzen,

Abstract

While XProc 1.0 is an XML-centric workflow language, XProc 3.0 attempts to overcome this restriction and to allow non-XML documents to flow between the steps of a pipeline. The intention for this change is not to make XProc a general purpose workflow language, but to enable pipeline authors to cover problems typically connected to XML processing. Although XProc 3.0 is still *work in progress*, this paper will present and comment its current concepts for processing non-XML documents. The main focus of this paper is to evaluate in a practical example, how a non-XML workflow which fits every day needs of pipeline authors can be developed in XProc 3.0 . The discussed use case deals with non-XML documents, that often appear in XML-related workflows such as zipped archives, ePUBs, images and JSON.

1. Introduction

XProc 1.0 [1] [126] is clearly an XML-centric language to design workflows, actually it is mostly an XML-only workflow language. To quote the XProc 1.0 specification:

> Although some steps can read and write non-XML resources, what flows between steps through input ports and output ports are *exclusively* XML documents or sequences of XML documents.

This XML-only approach proved to be fine for a lot of tasks, but as it turns out, even workflows dealing mostly with XML documents also have the need to deal with non-XML data. Just think of an ePUB mostly containing XHTML documents, but also having some JPEGs with illustrations, a manifest file which is pure text and finally being essentially a special kind of ZIP-document.

As such workflows show up quite often in real day life, the ability to deal with non-XML documents was a high priority requirement when developing the next version of XProc, which is called "XProc 3.0".[1]

In this paper I would like to give an introduction to workflows for non-XML documents in XProc 3.0. To do this as practically as possible I decided to layout a typical workflow involving the necessity to deal with non-XML documents and to show, how this could be done in XProc 3.0. Of course the workflow is a little bit of a made-up story because it was chosen for the purpose of demonstration. But it will show some basic structures of dealing with non-XML documents in XProc 3.0 and can serve as a blueprint for real life projects.

The workflow discussed here is this: We have a bunch of ePUBs in a folder somewhere and we have been asked to design a workflow which analyses the content of the ePUB and creates an RDF metadata description and an inventory in JSON which has to be sent to one of our inventory-servers which – for whatever reason – happens to understand only JSON. I will explain the details of this workflow later, but please keep in mind that it involves dealing with a lot of non-XML documents such as ZIP (the ePUB itself), plain text, graphics in JPEG, RDF and last but not least JSON.

This paper is divided into four parts, the third part being the central one:

- We will start with a short reminder on how one could deal with non-XML documents in XProc 1.0. As I said before, XProc 1.0 clearly is an XML-centric language, but there are some possibilities to deal with non-XML documents. To give a short reprise will hopefully help to understand the new features of XProc 3.0.

- In order to cope with non-XML documents, some fundamental changes had to be made in the transition from XProc 1.0 to the new XProc 3.0. The most important point here was to change the concept of a document that flows from one step to another. While in XProc 1.0 a document is a well-formed XML document only, for XProc 3.0 we needed a new document model which is able to cover XML and non-XML documents as well. In the second section of this paper we will take a short look at this new concept of a document as a foundation to understand how non-XML workflows can be created in XProc 3.0.

- The core of this paper is the third section where we will discuss in some detail how to design the sketched workflow for ePUBs. Here you will see the new document model in action and get to know some of the new XProc steps, which take advantage of this model and enable you to write workflows for non-XML documents.

- In the last section you will find a short summary and some conclusion concerning the suitability of XProc 3.0 for non-XML workflows.

A short caveat before we start: By the time of writing this article (May 2018), XProc 3.0 is still work in progress. While the document model can surely be seen as stable, the steps mentioned later are probably not. Though their basic outline is very unlikely to change, details may change as the discussion goes along. Especially the signature of the newly introduced steps and the dynamic errors raised by these steps might be subject to change in the process of standardising XProc 3.0. As a result of this situation, this paper is clearly not suitable as a tutorial on XProc 3.0, but is intended as a first look at the new possibilities in this language. Before you start to develop your own pipelines, please see http://xproc.org for the latest version of XProc 3.0.

2. Reprise: Non-XML documents in XProc 1.0

First a short reminder of what XProc 1.0 is, or I should say, was, because I believe pipeline authors will appreciate the new possibilities of XProc 3.0 so much, that they will switch to the new language version as soon as possible. One way to

---

[1]XProc 3.0 is currently developed by the XProc Next Community Group (https://www.w3.org/community/xproc-next/). The most recent version of the editor's draft is published on https://spec.xproc.org.

describe XProc 1.0 is to say that it is a pipeline language to design workflows *for* XML *in* XML. This definition is based on the fact, that only XML documents are allowed to flow between the steps that make up the pipeline. This feature of XProc 1.0 is most visible in an error message one gets to see from time to time when working with XProc 1.0:

> It is a dynamic error (XD0001) if a non-XML resource is produced on a step output or arrives on a step input.

Please note that this is dynamic error number 1. Although this error message is sometimes annoying, it clearly states the nature of XProc. XProc was invented and developed with the goal to be able to specify "a sequence of operations to be performed on a collection of XML input documents." Of course the creators of XProc knew that there are not only XML-documents, so there was a section on "Non-XML documents" in the original specs. This section is about thirteen lines and makes a distinction between almost-XML documents (HTML) and non-XML documents. For the first there is the ability to turn it into XML, and for the second a mechanism is presented, to let them "flow quietly through the pipeline": The non-XML document is either converted into text or base64-encoded and wrapped in an element node with a document-node, which means it now "can be processed like any other XML document".

This was of course an elegant way to deal with non-XML documents in XProc, but it also meant that these documents can only "go with the flow". XProc itself defines no steps for these documents, the only interesting things you could do with them is to send them to a web service or store them on your disk, provided they are not base64-encoded or your XProc processor implements the optional feature of decoding base64-encoded documents before it stores them. Of course you could store them anyway, but storing a base64-encoded JPEG wrapped in an XML element does not count as interesting, does it?

And as this situation became unpleasant, different mechanisms were invented to go around it. Steps were invented to deal with ZIP archives because we all know, that sometimes XML documents are packed into ZIP or should be packed into ZIP. But as ZIP documents could not flow through a pipeline themselves and although they may contain flowable XML-documents, they typically encapsulate a lot of stuff that could not flow with a pipeline. This is particularly problematic if you going to pack a ZIP, have a lot of XML-documents to put in, but also need some images and/or a text document.

To invent a workaround they probably looked at ANT and did what ANT does: Read and/or write to the file system. So unpacking a ZIP reads a file from the file system and creates a lot of files with the unpacked content on another place in the file system. Some steps just read from the file system but produce an XML-document that flows in the pipeline, e.g. XMLCalabash's extension step `cx:metadata-extractor` which reads an image and produces an XML-document containing the image's metadata. Other steps take XML-documents as input but write their non-XML result to the file system, like XProc's extension step `p:xsl-formatter` which typically produces PDF.

Some of these steps are really handy, some are mere workarounds. But from a purist perspective they are all workarounds because XProc is designed to be a language where documents flow between ports, and reading and/or writing to file systems is clashing with the style. But even if you do not share a puristic approach to language design, there are some problems with this approach:

- As your intermediate results are written to the file system, you have to remember to delete them, once they are no longer needed.

- Writing to the file system might sometimes also be dangerous because an incautiously designed pipeline could overwrite some (or even all) vital documents on your file system with its intermediate results.

- Sometimes it's not even handy, e.g. when you have to write a ZIP-archive to the file system and then read it again because you want to send it to some web service from the pipeline.

- From an implementer's perspective the read-/write-approach of steps has the disadvantage, that the connection between input- and output-ports is not the only thing to take into account, when the execution plan for a pipeline is to be determined. Since a step might need to read from the file system the results another wrote there, this establishes a connection between the steps too. I think this is one of the reasons why no implementation ever fulfilled XProc's promise yet, that a pipeline could be split up in parallel processes running side by side on our multi-core computers.

- Finally: Did I mention JSON yet?

3. XProc 3.0's new concept of a document

Before we look at the new document model introduced with XProc 3.0 in detail, let us first look at the challenges the XProc Working Group had to face in order to allow processing of non-XML documents. In XProc 1.0 everything that could possibly flow from the output port of one step to the input port of another step, was a well-formed XML document. And every step defined in the step library, if it has an input port at all, it has an XML input port. This situation

changes dramatically, once it was decided, that not only XML-documents could flow, but a wide variety of documents of different flavors. Then we have to differentiate between steps which could handle a certain kind of document, and steps, that have to raise an error because they could not possible do something useful with the document kind in question: Adding an attribute with a certain name and a certain value clearly only makes sense, when the document in question is an XML document. We do not even know what it could possibly mean to add an attribute to a text document or an image. Of course getting the image dimensions only makes sense of an image, but not for a text document or an XML document.

The other side of the same coin is a situation, when a pipeline author by accident exposes an image document to a step, that expects an XML document. Parsing an image document with an XML parser will most certainly lead into an error and the pipeline breaks. To be able to debug this kind of situation it is most useful not to get the information, that something is wrong with the XML document, as the parser will tell us, but to state, that a non-XML document appeared somewhere, where only an XML document is allowed. Thirdly, from the perspective of an XProc processor, it is important to know the kind of document it is dealing with, so it can choose an internal data model, that is suitable for the document type in question and for the operations defined for documents of this type.

The practical upshot of this reasoning is, that we need two things: We need to know what type a specific document has and we need a label for XProc steps, to say what kind(s) of documents they can deal with or allow on their specific input ports.

In order to cope with these two requirements, the XProc Working Group had to develop a completely new understanding of what flows on an XProc pipeline from step to step. What flows is still called "a document", but a document is now *a pair* consisting of *a representation* and the *document properties*. The representation is a processor specific data structure which is used to refer to the actual document content. The document properties are pairs of keys and values containing metadata of the content. The type or kind of the document flowing between the steps is the most important metadata and it is associated with the key `content-type`. XProc 3.0 uses the well known media type notation like `application/xml`, `text/plain`, `image/jpeg` and so on, to distinguish different types of documents.

Steps in XProc 3.0 now also use the media type notation to declare, which kind of documents are expected on a specific port. If a document that matches the step's specification arrives, everything is fine: The step can perform the expected operation and new documents (pairs of representation and document-properties) are produced on the step's output ports. But if an incoming document does not match the content-types expected by a step on a specific port, an specific error is raised by the processor, telling e.g. that step `add-attribute` expects a document with one of the media types `application/xml`, `text/xml` or `application/*+xml` but a document with content-type `text/plain` was found. Which content-types are expected on which ports is of course determined by the inner logic of the step: Steps like `p:identity` can obviously deal with any kind of document because it only passes through the documents appearing on this input port. The same is true for `p:count` which counts the number of documents on an input port and does not have to know, what kind of documents they are. But most steps known from the XProc 1.0 step library typically require a document with an XML media type to appear on its input ports, because adding an attribute, replacing an element, renaming a namespace etc. only makes sense for XML documents.

The new concept of a "document" in XProc 3.0 consisting of a representation of a content of a certain type and the document properties as its metadata as we will see nicely solves the problem of opening up the well known XProc conception to non-XML documents. Regardless of the type of the documents, they are produced by a step and exposed on one of its output ports. The XProc processor then sends them to another step's input ports and will raise an error, if the document's type does not belong to the types of documents the step in question is able to deal with. The processor is able to do this, because the document properties flow with the documents between the ports and so a step *knows* what type of document is coming in.

But what about the pipeline author, how is she able to access the document properties? You can easily imagine situations where a pipeline author might want to make a decision based on the type of the document, for example because the output of a certain step should be sent to step A if it is an image, but to step B if it is of some other type. To make this kind of processing possible, the document properties of a document are exposed to the pipeline author via a bunch of XPath functions making mostly use of the `map` type introduced with XPath 3.1 [2] [126]. More precisely the document properties are represented as `map(xs:QName, item())` and you can access them as a map by using the XProc extension function `p:document-properties($doc)`. Most of the time pipeline authors will not be interested in the full map, but want to retrieve a specific value. This can be done using `p:document-property($doc, $key)`. And finally there is a function called `p:document-properties-document($doc)` which returns an XML document for the document properties of the document in question. In this document each key of the map becomes an element and the value of the key becomes the element's values. In this way pipeline authors can retrieve and evaluate the document properties of a document using just familiar XPath expressions and do not need to use the new expressions introduced for maps and arrays.

Now that we have seen how the document properties of a document in XProc 3.0 are accessed the question may come up, how to control or set the document properties in a pipeline. Typically in most cases you do not have to do this,

because the XProc processor is responsible for this: If a step declares the documents on an output port to be, say of type `text/plain`, the document properties of the resulting documents are set by the processor accordingly. But sometimes obviously pipeline author need to control the document properties themselves, e.g. when you are loading a document and do not trust your file system to get the mime type of the document right. Another use case for setting document properties in a pipeline is when you create a document inline and want to tell the processor explicitly which document type the document has to have. For those cases XProc 3.0 provides an additional option named `document-properties` which takes a map with document's metadata as its values. Finally, if you need to add additional metadata to a document created by one step before it goes into another step, there is a new step called `p:p:set-properties`, which can be used to overwrite existing metadata or add additional data to the document properties.

Having talked about the new concept of a document being a pair of a representation and its document properties and having discussed the document properties to some extent, the next thing we have to cover is the representation part of the document pair. As said above, the representation is a data structure used by the processor to refer to the document's content. Which representation a processor has to use in order to deal with the content of the document is defined based on the document's media type. The current version of the specs (May-2018) calls out four different types of representations: Obviously there are *XML documents* identified by an XML media type, which is "`application/xml text/xml application/*+xml`". Secondly the specs mention *text documents* which are identified by media type "`text/*`". Thirdly there are *JSON documents* which have media type "`application/json`" and forth there are the so called *binary documents* which are identified by any media type not mentioned yet. Implementers are obviously free to implement additional document types identified by media types, as long as they do not conflict with the ones mentioned before.

As we learned before, the XProc 3.0 specification defines a *representation* for documents of a specific media type, where "representation" means "a data structure used by an XProc processor to refer to the actual document content". And this brings us to another important change, the XProc working group decided to make for XProc 3.0, because they say in the specs:

> Representations of XML documents are instances of the XQuery 1.0 and XPath 2.0 Data Model (XDM).

Therefore XProc 3.0 uses the same data model as other XML related technologies like XQuery or XSLT. And this is a strong change in the concept of what an XML document is. In XProc 1.0 the concept of a well-formed XML document was used, i.e. every document had exactly one element node, optionally preceded and/or followed by comment nodes, processing instructions and all whitespace text nodes. Well-formed XML documents are fine, but the stipulation, that every XML document has to be well-formed all the times is very burdensome when you try to make even very slight modifications. One thing I ran in a lot of times for one specific project, was the problem to add a processing-instruction as first node of the preamble, so the browser could recognize the XForms in the produced documents. This might sound like an easy task for someone familiar with XQuery or XSLT, but in XProc 1.0 it was tricky. The natural choice is of course `p:insert` where one matches the root element of the document, and tell the processor to insert the processing instruction before the root element. But you can not do this, because `p:insert` inserts *documents* into other documents, *not nodes*. So you can not insert a processing instruction, but have to insert either a processing instruction followed by a dummy element node or wrap the processing instruction into an element node in order to fulfill the "well-formed documents only" rule. But obviously you can not insert this document before the current element node of the document, because a well-formed document can not have two top level elements. So here is one way of doing this:

```
<p:wrap-sequence wrapper="dummy" />
<p:insert match="/dummy" position="first-child">
  <p:input port="insertion">
    <p:inline>
      <dummy2><?pi target?></dummy2>
    </p:inline>
  </p:input>
</p:insert>
<p:unwrap match="/dummy | /dummy/dummy2"/>
```

But in XProc 3.0 where an XML document is to be implemented according the XDM [3] [126] concept, one can do it in the most natural way:

```
<p:insert match="/" position="first-child">
  <p:with-input port="insertion">
    <?pi target?>
  </with-input>
<p:insertion>
```

The second document type, which is newly introduced with XProc 3.0, is a *text document*. Text documents are characterized by a media type that matches the scheme `text/*`, with the exception of `text/xml`, which is an XML document. Constructing a new text document is as easy as constructing an XML document:

```
<p:identity>
  <p:with-input>
    <p:inline content-type="text/plain"
      >This is a new text document</inline>
  </p:with-input>
</p:identity>
```

The XProc processor will produce a new text document on the output port of `p:identity`. This document will consist of a document node with just one text node child which holds the text. Doing the representation of text documents in this way has the obvious advantage that it fits perfectly with the use of XPath as an expression language in XProc. Suppose you have a sequence of text documents and for some reason you want to treat text document differently whose second word is "`is`". Since the text documents in XProc are a special kind of a document as defined in XDM, you can do it as easy as this:

```
<p:choose>
  <p:when test="tokenize(.,'/s')[2]='is'">
    <!-- ... -->
  </p:when>
  <!-- ... -->
 </p:choose>
```

To represent text documents as a text node wrapped into a document node also allows us to use them in `p:wrap-sequence` to wrap an element node around the text and thereby produce an XML document. Of course you can also use `p:insert` to insert the text node of a text document as a child of an already existing element node. And finally, if you select from an XML document and the resulting nodes are all text nodes, the XProc processor will create a new text document for you. We will see more applications for text documents when we come to discuss our example workflow in more detail, but for now we can record, that text documents in XProc 3.0 are pretty well integrated into the XML universe.

Next up is *JSON*. Integrating JSON into the XML world was a high priority during the last years and great work has been done to archieve this goal with the new standards of XDM 3.1 and XPath 3.1. If you take a look at XSLT 3.0 you will find that working with JSON feels almost as natural as working with XML. Based on the cited works, JSON is now also integrated into XProc 3.0. As you might expect from the preceding discussion, it is called a "`JSON document`". The document properties of a JSON document have a content-type entry that contains a JSON media type like "`application/json`". As JSON is a text based format, you can easily construct JSON documents within XProc pipelines:

```
<p:identity>
   <p:with-input>
     <p:inline content-type="application/json"
       >{"topic":["XProc", "3.0"]}<p:inline>
   </p:with-input>
 </p:identity>
```

As you might expect, if you are familiar with the treatment of JSON in XPath, the XProc processor will use the function `fn:parse-json()` on the string supplied and produce an XDM representation of this JSON document. In the given case it will obviously be a map item with one entry mapping a string to an array item containing two strings.

Now this representation is perfectly in accordance with what you might expect if you come from an XPath, XQuery or XSLT background, however it does not quite fit with the XProc concept of documents flowing between input and output ports. And this is because the representation of the JSON document is *not* an instance of an XDM node, but a map item (or an array item or an atomic value in other cases). And neither a map item nor an array or an atomic value can be the child of a document node per XDM. If you recall the definition of an XProc document you can now understand, why it is *not* defined as a pair of document properties and a node (which is true for XML and text documents), but as a pair of document properties and a *representation*. The representation for some document types *might be* an XDM node, but as for JSON documents it is not.

Let us take a closer look how this concept of a document fits into XProc and XPath. First of all, our JSON document produced on `p:identity` flows out of the step on an output port which is typically connected to the input port of

some other step. As said above, what happens then depends on whether the receiving step accepts a JSON document on the respective input port. For example if it is a `p:store` the document will be written to some destination as you might expect. But if the receiving step is e.g. a `p:add-attribute` a dynamic error will occur, because a JSON document is not allowed on the input port of this step. But this is nothing special for JSON documents but applies to all documents in XProc. If, for example, an XML document appears on an input port that only allows JSON documents to flow in, a dynamic error is raised too.

As you can see, JSON documents are first class citizens in XProc 3.0 when it comes to the question of what can flow between steps. But if you are familiar with XProc, you might recall, that documents do not only flow between steps, but can also appear as context items when it comes to evaluating XPath expressions. Here JSON documents do not fit quite as well, because their content is not represented as something which is an XDM node and therefore an XPath expression like `"/"` can not expose the content to XPath. For `"/"` the XProc processor is required to construct an empty document node, so `p:document-properties('/')` will return the document properties of JSON documents as well. To overcome this problem is obviously very easy if you imagine an XProc defined XPath function, which takes the document node associated with the JSON document as a parameter and returns the same representation of the JSON document as XPath's `fn:json-doc()` would. As of May 2018 you will not find such a function in the specifications for XProc 3.0, but I am pretty sure the community group now taking care of XProc's development will find some way to bridge the gap between JSON documents and XPath expressions.

Finally XProc 3.0 defines a fourth document type called "*binary document*". A binary document is actually *anything* which is not either an XML document, or a text document or a JSON document, or, more precisely which has a media type not associated with these three document types. This document type sums up such different kinds of data as ZIP-archives, all kinds of images, PDF-documents and every thing else we have on our file systems or receive from web services. As for JSON documents the XProc processor is required to construct an empty document node, so `p:document-properties('/')` will return the document properties associated with this document. How a binary document is represented internally by an XProc processor is *implementation defined*. And it is obvious that not all binary documents will be internally represented in the same way by an advanced XProc processor: Smaller documents will probably be held in memory for fast access, but if you think about very large documents (as a video or an audio file), some optimization will be necessary. One strategy is to store those files away in a temporary folder and let just references to these files flow between the steps. Only if a step actually needs to access the document's content, the file or parts of it are loaded by the XProc processor.

Because the representation of binary documents has to be implementation defined, XProc 3.0 currently defines no way to access the document's content within an XPath expression. One can easily imagine an XProc defined XPath function returning the document's content as `xs:base64Binary` or as `xs:hexBinary`. But the main problem here is, that in most cases you do not want the whole document content, but are only interested in a smart portion of it. For this reason an implementation returning the whole content, which may be very large, and then use XPath expressions to identify the small range the pipeline author is really interested in, would be very inefficient. This problem is not impossible to solve, but the XProc community has not agreed on a solution yet. One way to solve it would to determine the content's size, either as part of the metadata in the document-properties or as a function taking the binary document as argument. This might be complemented by an XProc defined XPath function which allows to select a part of the document's content. The specification of the EXPath binary module could certainly be a role model for solving this problem.[Kosek:Lumley:Binary:Module:1.0]

Together with the new document model, XProc 3.0 introduces a new step to convert or cast the different document types into each other: `p:cast-content-type`. This step takes an arbitrary document on its input port and the content-type this document should be casted to and returns a casted document on the output port (or throws an error if the XProc processor is not able to perform the requested casting). This abstract characterization is necessary, because this step is a kind of "Jack of all trades" of document processing in XProc. The easiest task this step can perform, is to cast from one XML media type to another, say from "`application/xslt+xml`" to "`application/xml`" or vice versa. Here the actual document representation does not need to be changed in any way, just the value of key `content-type` in the document properties needs to be changed. Casting from a non-XML document type to an XML document type will produce an XML document by wrapping the representation of the non-XML document into a `c:data`-element. This type of casting is well known from XProc 1.0, where the element `p:data` on an input port was responsible for converting non-XML to XML.

The step `p:cast-content-type` can also perform the opposite casting from an XML document with a `c:data`-element with encoded data as a child to the respective document type. All other conversions between media types are currently implementation defined. In this area some more work needs to be done, for example when it comes to cast a JSON document to an XML document. The current version (May-2018) of the XProc 3.0 specification defines that is has to result in a `c:data`-document with a base64-encoded representation of the JSON content. In *XPath and XQuery Functions and Operators 3.1*[2] [126] we find a mapping from JSON to XML which is used in the two functions `fn:json-to-xml()` and `fn:xml-to-json()`. Making use of this mapping when it comes to casting a JSON

document to an XML document and vice versa in XProc 3.0 is certainly an idea that should be discussed. In this line of thought we might also have a mapping from text document with media type "`text/csv`" to an XML document and vice versa. But some of the possible casting tasks to be performed by `p:cast-content-type` could definitively be scary. Let me just mention the case when an XML document with media type "`image/svg+xml`" should be casted to "`image/jpeg`".

This much on the new concepts (and steps) introduced by XProc 3.0 to escape the XML-only limitation and to allow the design of XProc workflows for non-XML documents. Let us now come back to the use case shortly introduced at the beginning of this paper and discover the practical aspects of non-XML workflows in XProc 3.0.

4. Applying the model

The non-XML workflow to be developed here was shortly introduced above, but now let us look into details and see how we could realize it in XProc 3.0. As you might recall, the workflow deals with ePUBs stored somewhere on our file system. And our workflow should create some RDF metadata about the ePUB's content and create an inventory which has to be sent to a JSON-only web service. As said above this workflow is a made-up story to explore the new possibilities of XProc 3.0. It is not a real life project, but could serve as a blueprint for those.

First let us sum up, what kinds of non-XML documents are involved in our workflow: First of course we have ePUBs which are essentially ZIP documents with a defined structure. Then we have to produce some metadata according to the RDF model, which might be represented as XML (like in RDF/XML or RDFa). But as we deal with non-XML document types, we will of course use one of the text based serializations of RDF, namely Turtle. The source for our metadata generation will be the Dublin core metadata information expressed in the ePUB's root file. Our ePUB will typically also contain a lot of image files and someone wants to know, which images are used in which ePUB. So we will have to create an inventory of all the images (JPEG, PNG and GIF) in the ePUBs together with their width and height. This inventory has to be in JSON because we need to send it to an inventory server which only understands JSON. So we have a pretty zoo of different non-XML document formats to deal with in our pipeline.

Let us start with the outermost step which has just the task of finding all ePUBs in a given folder:[2]

```
<p:declare-step version="3.0">
  <p:option name="epub-folder" as="xs:anyURI" required="true" />

  <p:directory-list path="{$epub-folder}"
      include-filter=".*\.epub"
      recursive="true" />

  <p:for-each>
    <p:with-input select="//c:file" />
    <epp:analyze-epub>
      <p:with-option name="href"
          select="concat($epub-folder,/c:file/@name)" />
    </epp:analyze-epub>
  </p:for-each>
</p:declare-step>
```

For those readers with little or no experience in XProc, let me just say that the step `p:directory-list` will produce a list of content for the directory specified by `path`, in our case containing only directory entries which match the regular expression given with `include-filter`. The step produces an XML document with a `c:directory` root element containing `c:file` or `c:directory` elements. Since we are only interested in (ePUB-)files, the `p:for-each` will select all the respective elements. The treatment of the ePUB is actually done in the user defined step `epp:analyze-epub`, which is called with the ePUB's absolute URI as option value.

Readers familiar with XProc will discover some of the new features of XProc 3.0 in this example: We now have typed values for options and variables (expressed by attribute `as` on the `p:option`-declaration). While in XProc 1.0 all values of options or variables were either strings or `untyped atomic`, they can now have any value (including documents, nodes, maps and arrays) and the XProc processor has to make sure they only have a value of the declared type. The second point you might have discovered are the curly braces and if you are familiar with XSLT just might have guessed that they are attribute value templates. If so, you are right. XProc 3.0 introduces attribute value templates and text value templates (as known from XSLT 3.0). AVTs prove to be very handy to write shorter pipelines, because now you can use the attribute shortcut for options even if they contain XPath expressions. The long form with `p:with-option` will

---

[2]To keep the following code readable, I will omit all namespace declarations.

only be necessary if your XPath expression refers to the context item because the context item for AVTs is undefined. This is why we have to use the explicit form in supplying the value for `href` on `epp:analyze-epub`. And finally you may have discovered that `p:directory-list` has a new attribute, so you can decide whether you only want a listing for the top level directory or also for any directory contained.[3] So this pipeline might have some interesting new stuff, but when it comes to non-XML documents it is quite boring. So let us go on and see how we can deal with non-XML documents.

The first non-XML format we have to deal with is of course ePUB which is a special kind of ZIP archive. Uncompressing and compressing this kind of archive format is already essential to many traditional XProc workflows, because it is not only needed for ePUBs but is also the underlying format for "docx". Using the framework of XProc 1.0 two different approaches have been developed to deal with archives: The step `pxp:unzip`, proposed by the EXProc-community allows you to extract one document which will appear on the step's output port: If the document has an XML context-type, the document itself appears, but for every other document a `c:data` document is returned which contains the base64-encoded representation of the selected zip's content. This approach is totally in line with XProc's basic concept, but obviously has a lot of limitations. The second approach to deal with ZIP archives is represented by the step `tr:unzip` which is part of the `transpect`-framework developed by le-tex publishing services.[4] This step extracts a complete ZIP archive (or a single entry) to a specified destination folder in the file system. Here the XML-only limitation is circumvent by writing to the file system instead of exposing the base64-encoded content on a result port. But it obviously breaks away from the concept of documents following between steps on ports.

In XProc 3.0 we can now have the best of the two approaches: Thanks to the extension of the document model now XML and non-XML documents can flow on the output ports of a new `uncompress` step, which might have the following signature:[5]

```
<p:declare-step type="xpc:uncompress">
  <p:input port="source" content-types="*/*" />
  <p:output port="manifest" sequence="true"/>
  <p:output port="result" content-types="*/*"
      sequence="true" primary="true"/>
  <p:option name="include-filter" as="xs:string+" />
  <p:option name="exclude-filter" as="xs:string+" />
  <p:option name="method" as="xs:token" select="'zip'" />
</p:declare-step>
```

The ZIP-archive flows into this step on the port `source` which intentionally accepts all content types. The first reason is that ZIP archives can appear with many different media types where some do not even have the suffix "zip". The second reason is, that this step is designed as a kind of Swiss knife for different kinds of archive formats. The documents contained in the archive flow out on the port `result`, which is the primary output port. For a typical archive this will be a sequence of different document types, where each document is a pair of a representation and its document properties. The options `include-filter` and `exclude-filter` can be used to control, which entries from the archive should appear on the output port. Like the options with the same names used on XProc's standard step `p:directory-list` they are interpreted as regular expressions used to match the names of the archives entries. Unlike its predecessor the step now make use of XProc's new alignment to the XDM type universe. So we can now supply a sequence of regular expressions and say, that an archive's entry is returned on the output port, if its name is matched by at least one of the regular expressions on `include-filter` and by none of the regular expressions of `exclude-filter`. Obviously this gives you a very powerful mechanism to control, which entries are extracted from the archive and which are not.

Now let us put this step into action in the workflow we have to design. We are not interested in all archive entries, but only in the image files (since we have to create an inventory of them) and the root file, since it contains the metadata we are after. For brevity I will skip the problem of identifying the ePUB's root file by inspecting entry "META-INF/container.xml". We will guess, that the root file is found in a document named "package.opf". Also for brevity the following pipeline will read the ePUB twice, once to extract the root file and another time to extract the graphic files. In a real life project one would probably only open the ePUB once for efficiency reasons and then split the resulting sequence into the root file and the graphic files. Here is what our step `epp:analyze-epub` might look like:

```
<p:declare-step type="epp:analyze-epub">
  <p:option name="href" as="xs:anyURI" required="true" />
```

---

[3] As of May 2018 you will not find this option in the specification, as we have not done much work on the standard step library yet. There was a request for this feature which I think is very handy, but the name of the option and its exact behaviour can not be taken for granted yet.
[4] See http://transpect.github.io/modules-unzip-extension.html.
[5] Again, this exact specification of this step is not formalized in the specification yet. It is pretty sure that such a step will be part of XProc 3.0's standard step library, but the exact signature (e.g. names and types of the options) and the step's exact behaviour is still under discussion.

```
<xpc:uncompress include-filter=".*/package\.opf">
  <p:with-input href="{$href}" />
</xpc:uncompress>
<epp:extract-metadata />

<xpc:uncompress>
  <p:with-input href="{$href}" />
  <p:with-option name="include-filter"
    select="('.*\.jpg', '.*\.png', '.*\.gif')" />
</xpc:uncompress>
<epp:create-inventory />
</p:declare-step>
```

If you look at this short pipeline, I think you will recognize how natural it is now to work with non-XML documents in XProc 3.0. We extract an XML document named "package.opf" from the ePUB and let it flow into step `epp:extract-metadata` and we extract the relevant graphic files from the ePUB and let a sequence of non-XML document flow into step `epp:create-inventory`.

Now let us turn to the conversion of the Dublin core metadata contained in `opf:metadata` of the ePUB's root file into the Turtle serialization of an RDF graph. By looking at the ePUB's root file, we will find the metadata as the following example shows:

```
<opf:metadata
    xmlns:opf="http://www.idpf.org/2007/opf"
    xmlns:dc="http://purl.org/dc/elements/1.1/">
  <dc:identifier>urn:isbn:978-80-906259-2-1</dc:identifier>
  <dc:title>XML Prague 2017</dc:title>
  <dc:language>en</dc:language>
  <dc:creator>Jiří Kosek</dc:creator>
  <dc:date>2017</dc:date>
</opf:metadata>
```

From this format we need to generate a Turtle serialization which should look like this:

```
<urn:isbn:978-80-906259-2-1>
  dc:title "XML Prague 2017" ;
  dc:language "en" ;
  dc:creator "Jiří Kosek" ;
  dc:date "2017" .
```

I think the first intuition of many readers will be to use XSLT for this conversion. As an XProc author I would definitely agree with this intuition and write an XSLT stylesheet called by XProc's `p:xslt` to invoke the transformation. With XProc 3.0 this is possible because the text document created by XSLT is now a first class citizen of a pipeline and there is no need anymore to wrap it in an element node to make it an XML document.

But as this paper deals with non-XML documents in XProc 3.0, let us see, how this could be done without invoking an XSLT transformation. The following fragment shows how one might do it:

```
<p:variable name="id" select="/opf:metadata/dc:identifier" />
<p:for-each>
  <p:with-input select="/opf:metadata/dc:*[not(name(.) =
    'dc:identifier')]" />
  <p:variable name="entry" as="document-node()" select="." />
  <p:identity>
    <p:with-input>
      <p:inline content-type="text/turtle"
        >{$entry/*/name()} "{$entry/*/text()}"</p:inline>
    </p:with-input>
  </p:identity>
</p:for-each>
<xpc:aggregate-text separator=" ; &#xD;" />
<xpc:add-text text="&lt;{$id}&gt; &#xD;" position="before" />
<xpc:add-text text="." position="after" />
```

Here a text value template (known from XSLT) is used to create a text document for every element entry in the Dublic core namespace. We have to use the variable `entry` (which is a document node), because as for AVTs the context item is undefined for TVTs. What appears on the output port of `p:for-each` is a sequence of text documents, each containing the predicate and the object of a statement. The step `xpc:aggregate-text` then takes this sequence of text documents to create one single text document. Between two adjacent text documents a semicolon and a carriage return is inserted. And finally the two appearances of `xpc:add-text` put the ePUB's identifier in front of text and a colon behind it so we have a valid Turtle statement.

As you can see, text based format like Turtle can be very easily created using the new features introduced with XProc 3.0. Currently the standard step library does not contain any steps dealing especially with text documents. The two steps used in the previous example are pretty good candidates for this, but I am not sure they will make it into the final library. The reason here is, that both steps can be written in XProc itself using the string functions provided by XPath. So it might be a question of principle whether to include such steps in the standard library which would make pipeline authoring more convenient or ask the authors to import their XProc implementation into their pipeline every time they need this functionality.

When it comes to RDF itself as a theoretical concept, there is currently no support in XProc 3.0. Of course RDF/XML and RDFa are supported as they are XML documents and text based serialization formats of a graph can be handled as we have just seen. But an RDF graph as a theoretical concept in opposition to its various representations is currently not one of XProc's document types. The XProc Next Community Group has mentioned RDF several times in their discussions at the various meetings, but not really tackled the topic yet.[6] There might be good reasons to extend the current document model by RDF graphs. The RDF document type would be independent of any serialization form and there could be steps to parse a serialization form to an abstract graph and to serialize the graph. Additionally we could have steps, that add triples to a graph or remove a specific triple etc. Going further one could wish for a step to validate the graph with SHACL or another step to query the graph with SPARQL. In his paper for XML Prague 2018 Hans-Jürgen Rennau [5] [126] argues that XML and RDF are complementary concepts and that "an integration of RDF and XML technologies is a very promising goal." Given our previous discussion one might think of XProc as one of the places where this integration might take place. But as I said before, there is no decision on whether RDF graphs will become part of XProc's document model. And there might be doubts they will make it, because sometimes its better to get things done, than to get things perfect.

Let us go back to our workflow example. Having dealt with ZIP archives and ePUBs, text documents and Turtle we now have to turn to the last open point of our workflow which is to create a JSON inventory of the image files contained in the ePUB. Given the fact one of XProc's mayor use cases today is in publishing, the lack of support for images and image processing is surely striking. Pipeline authors had to step in here and write their own extension steps to do at least some rudimentary image processing.[7] But this is typically only an in house solution because you have to write these steps in the programming language the XProc processor used is written in and you have to make known these steps to the processor in some vendor specific way. Pipelines using these steps are not interoperable with other XProc processors or other configurations of the same processor.[8]

As we saw above, XProc 3.0 changes this with the introduction of the new document model which allows images to be loaded in a pipeline and to flow between steps. Currently there are no special steps dealing with images, but you can easily imagine steps that extract data from an image document or do some image processing e.g. scaling. And finally it is now very easy to create an ePUB containing XML documents and images alike. The old workaround was to create an intermediate folder on the file system, storing all XML documents into this folder and copying all the images there too, and then call a step to create an archive from the respective folder. With the new document model you will not need this workaround anymore but can simply have a zipping step, taking all the (XML and non-XML) documents on its input port sequence and creating an archive from it to appear on the output port.

As you might recall, the workflow we are designing, involves some image processing because we are requested to create an inventory of all the image files contained in an ePUB and this inventory has to contain the dimensions of the images as well. For this we need a step that takes an image document on its input port and produces an XML document containing the required information on its output port. The signature of such a step might look like this:

```
<p:declare-step type="xpc:image-profile">
  <p:input port="source"
    content-types="image/jpeg image/png image/gif" />
  <p:output port="result" />
</p:declare-step>
```

---

[6]For XProc 1.0 there are some attempts to deal with RDF documents. The most prominent are, of course, the RDF extension steps for XMLCalabash.[4] [126]

[7]See for example the steps `image-identify` and `image-transform` from le-tex's transpect framework.

[8]See [6] [126], especially section 5.

On the step's output port an XML document appears containing information about the image, which might look like this:

```
<c:image-profile>
  <c:image-property name="name" value="pic1.jpg" />
  <c:image-property name="mimetype" value="image/jpeg" />
  <c:image-property name="width" value="300" unit="px" />
  <c:image-property name="height" value="500" unit="px" />
  <!-- more properties to come here -->
</c:image-profile>
```

The XProc Next Community Group has not decided yet about the format of the resulting XML document. As for some applications the use of attributes seems to be convenient, other applications may prefer to have the properties as element names and the values as text children of these elements. There is no reason why such a step should not have an option allowing the pipeline author to select between these and other possible formats. Actually for the workflow discussed in this paper it would be very handy, if the output is not restricted to different varieties of XML documents, but could also be a JSON document, as we have to send the graphics inventory to a JSON only web service.

So it comes to JSON as the last data format or document type we have to consider in our workflow. From what we have done so far, we could have an XML document containing all the `c:image-profile` documents for the image files of an ePUB.[9] And there are different ways to produce the lexical JSON we would like to send to a web service:

The first way to produce a JSON representation of this document has little to do with the newly introduced JSON document type in XProc, but uses text documents as a vehicle for lexical JSON. And of course it makes use of the XPath function `fn:xml-to-json()` which takes an XML document in a special designed vocabulary as an argument and returns a string conforming to the JSON grammar. Since we need a textual representation of the JSON document if we want to send it to a web service, the string result here is fine for us. If we would need actual JSON, calling the function `fn:parse-json()` with the previous function call's result as a parameter would do the job. All we need to do to generate the JSON document therefore is (1) call an XSLT stylesheet that takes our source document and transforms it into the XML format expected for `fn:xml-to-json()` and (2) create the request document for `p:http-request`.

The second possible strategy to create a lexical JSON representation of our image inventory document is to create the text document directly with XSLT without the intermediate step of creating an XML-document in a format suitable for calling `fn:xml-to-json()`. This might be a plausible strategy too, but as XML to XML transformation with XSLT is an everyday job, one might be better off doing this and leaving the problem of transformation to JSON to the processor's built-in function.

Thirdly we could create the lexical JSON document directly in XProc as we have done with Turtle in the example above. Lexical JSON and Turtle are both text based formats so using XProc 3.0's new text documents seems to be a practicable way.

Taking all these possibilities together, one might come up with the question if the JSON document type introduced with XProc 3.0 has a meaningful purpose at all.[10] The underlying impression is boosted by the fact, that the only step currently defined for JSON documents is `p:load`. One might expect, that one processor or the other additionally may support JSON documents in `p:store` (as non-XML serialization is an optional feature). As no other step is currently defined in XProc's standard library one has either to rely on the processor or site specific extension steps or (as I would expect) convert JSON to XML (`fn:json-to-xml()`) and back (`fn:xml-to-json()`).[11] This shortcoming may in part be due to the pending update of the step library. For example: XSLT 3.0 widened the concept of the "initial context node" to the new concept of the "initial match selection", which includes not only a sequence of documents, but also a sequence of parentless items like (XDM) values and maps. This change in the underlying technology will most certainly be reflected in an updated signature of `p:xslt`.[12] And this updated signature might also allow JSON documents to flow into the input port of `p:xslt`. Along this line of thinking `p:xquery` might be another step where JSON documents flow in (and out).

Another way to make JSON documents more useful to pipeline authors may be the introduction of JSON specific steps into XProc 3.0's standard step library. Concerning our task to create a JSON document from the sequence of

---

[9]For brevity the XProc snippet is left out here: It is just a `p:for-each` iteration over the image documents delivered from `uncompress`, calling `xpc:image-profile` on each and do a `p:wrap-sequence` on the sequence flowing out of the `p:for-each`.

[10]To avoid misunderstanding: We are talking about JSON documents as an XProc document type not about maps and arrays as part of XDM. Having variables and options that can contain maps and arrays is useful without doubt. Replacing parameter ports with maps should count as a prove.

[11]See the above discussion on implementation defined aspects of `p:cast-content-type`.

[12]To ensure compatibility with legacy pipelines or to allow the use of XSLT 2.0-only processors, one might also think of adding a new step for XSLT 3.0 transformation.

XML documents with image information, a step that creates a JSON document containing a map and a step that joins a sequence of JSON documents with maps to one single JSON document would be helpful. Omitting the exact specification of such steps, a possible pipeline might look like this:

```
<p:for-each>
  <p:output port="json-info" content-type="application/json" />
    <p:variable name="props" select="[
      xs:string(//*[@name='mimetype']/@value),
      xs:string(//*[@name='width']/@value),
      xs:string(//*[@name='height']/@value
    ]" />
  <xpc:json-document>
    <p:with-option name="value" select="
      map{xs:string(//*[@name='name']/@value) : $props}
    "/>
  </xpc:json-document>
</p:for-each>
<xpc:aggregate-json-map />
```

The result here should be a JSON document containing a map, where in each map entry the key corresponds to the name of the image file and the value of each entry is an array containing the mime type, the width and the height as strings in that order. Obviously this might be done in a more elegant way, but as we are concerned only with the basic concepts here, this example might be sufficient.

Thinking along these lines we might invent other JSON specific steps which might be useful in pipelines. If we restrict our selves to JSON documents that are maps, one might think about a step, that adds one entry to the map:

```
<p:declare-step type="xpc:add-to-json-map">
  <p:input port="source" content-types="application/json" />
  <p:option name="key" as="xs:string" required="true" />
  <p:option name="value" required="true" />
</p:declare-step>
```

And then of course we would also need a step to remove a key/value entry from a map e.g. by giving a key.

But the key problem to me with JSON documents still is their connection to XPath and XDM which is, as I said above, currently missing in XProc 3.0: While we are able to express something like "remove the third child element of this node in this XML document" for XML documents, we are not able to say "remove the third key/value pair from the map in this JSON document". We might invent some steps for JSON documents, but currently we are limited to mutations of the top level map (or array) since we are not able to say something like "select entry four of the array that is associated with key a".

The other problem of course is, that JSON and JSON documents can not be mapped to XDM instances on a 1:1 basis: This is because a JSON document (at its "root") may either be a map or an array or an atomic value. Therefore the above proposed step `xpc:add-to-json-map` is quite naive because it presupposes, that the JSON document on the source port contains a map. But if the top level object of this document is not a map, an error would be raised most probably. And this error could not be avoided by a prior checking, because currently there is no way to ask, whether the top level object of a JSON document is a map or something else.

To sum up our discussion of JSON documents I think it is fair to say, that some more work has to be done, to make them really useful to pipeline authors. This (preliminary) assessment is surely contrasted by the fact, that we can do a lot of useful things with JSON in XProc 3.0, because we now have maps and arrays for variables and options, and we have text documents for lexical JSON. Finally with the XML representation of JSON invented for XPath 3.1 we have a lossless way of representing JSON in XML, can make use of all the XProc steps to manipulate the document and then put it back to lexical JSON in order to store it or send it to the web.

5. Conclusions

Opening up XProc to enable processing of non-XML documents is one of the primary goals in the currently ongoing development of XProc 3.0. In this paper we took a first look at the new document model and investigated its application in non-XML workflows. While XProc still models workflows using the idea of documents flowing between steps from output to input ports, the concept of a document changes in the new version of the language. We found, that the new understanding of a document as a pair of a representation and document properties fits in perfectly into the well known world of XML, XPath and XDM.

With regard to the different instances of the new document model already defined in the language's specification it was argued, that using the XDM concept for XML documents makes pipeline authoring much easier. The same holds for text documents, which very smoothly opens up XProc to the processing of character based documents using a (not yet complete) set of XProc steps and XPath functions as well. The introduction of the binary document type allows processing of many document formats typically occurring in XML workflows, namely ZIP based formats as ePUBs and documents from various office productivity software suites on one side and image files on the other side.

Our discussion also showed some open questions which should be discussed in the further development of XProc 3.0: While XML based and text based serializations of RDF graphs are supported, there is currently no support for RDF graph as document type. Whether this would improve the work of pipeline authors in such a way that its introduction could be justified, has to be taken under investigation. Our coverage also showed some open questions about the usefulness of the JSON document type, which should be settled in further discussions.

Bibliography

[1] *XProc. An XML Pipeline Language*. 11th May 2010. Norman Walsh, Alex Milowski, and Henry S. Thompson. World Wide Web Consortium (W3C). http://www.w3.org/TR/xproc/.

[2] *XPath and XQuery Functions and Operators 3.1*. 21 March 2017. Michael Kay. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-functions-31/.

[3] *XQuery and XPath Data Model 3.1*. 21 March 2017. Norman Walsh, John Snelson, and Andrew Coleman. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-datamodel-31/.

[4] Norman Walsh. *XML Calabash Reference*. 06 March 2018. http://xmlcalabash.com/docs/reference/.

[5] Hans-Juergen Rennau. *Combining graph and tree: writing SHAX, obtaining SHACL, XSD and more*. Jiří Kosek. XML Prague 2018. February 8–10, 2018. Prague, Czech Republic. . 2018. 107-135. http://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf.

[6] Achim Berndzen and Gerrit Imsieke. *Interoperability of XProc pipelines*. A real world publishing scenario. Charles Foster. XML London 2016. June 4–5, 2016. London, United Kingdom. . 2016. 82-98. doi:10.14337/XMLLondon16.Berndzen01.

Lightweight XML DevOps using Apache Ant

Philip Hodder

Abstract

Many teams struggle with defining, documenting and following proper processes. A case study is described where the content delivery team of a major digital publisher was experiencing these problems when running transformation pipelines using XSLT and Schematron, as well as numerous other XML related tasks. The use of the Apache Ant build tool has allowed them to create a set of higher level tools that to enforce an agreed development policy using a set of reusable macros. These macros have allowed the team to dramatically increase the quality of deployed code, as well as improving the ability of new team members to become productive more quickly. This experience has spurred the development of an open source equivalent of this macro library (whimsically entitled "XPantS") which is also described.

1. Introduction

This paper presents a case study on the development of a set of standard tools to support and enhance XML based development practices in a content delivery team in a major publisher of digital information. Originally the team used Microsoft's Team Foundation Server (TFS) as a code repository and Apache Ant [https://ant.apache.org] as a build tool to perform a variety of XML related tasks, principally XSLT and Schematron [http://schematron.com] development for a wide variety of types of content with numerous internal end users. While TFS is a functional tool it does have certain limitations (branching, for example, among others) but was not being used in an optimal fashion: the entire code base for all projects in all the content teams was included in a single repository. Build processes were also poorly documented —as was the code and the build files themselves. Deliverable XSLT code was built using a complex web of Ant "includes" with temporary changes being made to certain of these build files by individual developers to build the code they needed to deploy at that time. Another developer would then make similar modifications to their copy for their work. Even worse, the transformation pipelines that converted content often ran the code directly from this repository so there was no separation between code in a development state and code being used in a production environment.

In summary then the team was having to work around a number of issues:

- Poor (or non-existent) documentation, leading to long on-boarding times before new team members (or existing team members changing roles) could become productive.
- No structured development process, which also led to team members unfamiliar with a new project or pipeline taking increased time to get useful work done.
- Poor repeatability of the build process and mechanisms.
- Lack of a formal (or even notional) deployment phase, leading to confusion over which version of code was supposed to be run and when.

Although it was not referred to within the team in these terms, these seem to be fairly standard "DevOps" issues, and it was becoming increasingly clear that improvements would have to be made.

2. Development Policy

The first change to be implemented was the agreement of a "development policy", a document that described what would become the team's set of "best practices". Although this was not a piece of technical work it did serve as a high level requirements definition for the macro development work.

The development policy included statements concerning:

- When, where and how to document the various kinds of projects and pipelines that the team worked on. Principally this consisted of guidelines to embed documentation within the various build files, XSLTs and Schematron files themselves.
- A directive to move to Git as the version control system with VSTS as the repository.
- Branch naming conventions (mostly following Git Flow, although that process was not used itself).
- Guidance on common naming conventions for folders within repositories to make transferring to new projects easier.
- Use of Semantic Versioning [https://semver.org] in projects and when to bump major, minor and patch numbers.
- Deployment rules, such as only deploying from a "release" branch, not deploying with outstanding commits etc.
- Use of pull requests to ensure at least some degree of code review and confirmation that documentation had been completed.
- Setting up an environment separate from other teams where the content delivery team could deploy assets (the "Content Architecture Production Environment" or "CAPE").

2.1. Documentation

A note on documentation: early on the decision was taken to use Markdown for all documentation within the team. While XML document standards like DocBook [https://docbook.org] and OxygenDoc would seem to be the obvious choice, Markdown [https://daringfireball.net/projects/markdown/syntax] was chosen for the reason that almost no training is required to use it—in fact merely using plain text will be accepted as valid and suitably rendered. This would reduce the workload on the content delivery team who would not have to learn another documentation standard, the assumption being that any form of documentation is preferable to no documentation at all. It should also be noted that Ant documentation cannot use these other (XML) standards as no tags are permitted inside its `<description>` tag —using Markdown therefore allowed the team to use a uniform documentation system. However, as part of the library described below some XSLTs were created to extract the Markdown documentation from Ant build files, XSLTs and Schematrons which could then be used to produce HTML for deployment.

3. The ANT macro library

It was clear that continuing to use the existing build process was unsustainable, especially as the team was growing and required its members to be able to rapidly switch roles within it. Therefore, using the Development Policy as the basis for the team's requirements, a new Ant macro library was developed.

Although there are many build systems available it was decided to keep using Apache Ant for two main reasons. Firstly, it is an XML based technology which would be familiar to the team, and secondly it would allow the team to re-use parts of the existing build system where appropriate. The various Ant macros in the existing build system were analysed for common code and functionality and a new library constructed using the following principles:

- Macro definitions would be used rather than custom Java code. This would be simpler to maintain and would not require additional skills beyond XML.
- Modular code: related macro definitions would be grouped together as files/modules to minimize the number of `<include>` statements required.
- Documentation for each of the modules would be included in the module itself (as Markdown in the `<description>` tag, as described above) and automatically generated for deployment to an internal web site.
- Common sense naming conventions were followed: for example, the file "compile-schematron.xml" contains macros involved in Schematron compilation and so on.
- Performance of the build system was not considered to be an issue; instead maintainability of the code by the whole team was considered to be more important.
- Additional configuration tools like Ivy [http://ant.apache.org/ivy/] and Maven [https://maven.apache.org] were also not involved, again for simplicity and maintainability reasons.

The macros in the library were developed over time but can be grouped into a few broad categories:

- XML operations, such as compiling Schematron files, validating XML instance documents and so on.
- Deployment of files using a variety of mechanisms.
- Wrappers around shell commands/system executables, such as *git*, *ssh* and so on.
- Utility macros used by other macros.

A number of XSLTs were also developed to work with these macros. These generally involved:

- Extracting Markdown documentation from Ant build files, XSLTs and Schematron.
- Filtering and converting SVRL reports.
- Modifying Schematron files in a number if ways, for example to display report and assert IDs in their respective messages.
- General utilty stylesheets to modify XML documents in various ways (remove documentation elements, linearise XML files etc).

These XSLTs were deployed to a standard location on the CAPE where they could be referenced by the Ant macros.

It should be noted that most these macros are essentially "wrappers" around existing Ant functionality—but the value of the Ant library lies in this encapsulation of higher level functionality. For example, the `deploy-files` macro is really just a wrapper around the Ant `<copy>` task. However, `deploy-files` is aware of the context within which the team is working: team members merely supply a target path on the CAPE and a relevant file set and the macro does the rest.

This "wrapper" pattern has proved very useful in day to day use. For example, the standard deploy target for the CAPE is an anonymous Samba share hosted on a AWS EC2 Linux instance. This allows Ant's basic `<copy>` mechanism (that works over file systems) to be used for deployment, as well as allowing developers to browse the deployment to check for correctness. However, a recently Windows 10 patch removed the ability for Windows 10 machines to connect to anonymous shares, which meant that those developers could no longer deploy any files. To get around this the `deploy-files` macro was updated to use optionally use *ssh* (Secure Shell) as a deployment mechanism. Once the relevant *ssh* keys were installed and configured they could carry on deploying assets with no changes required to their existing build files.

3.1. Example of use

The usefulness of this library to the content delivery team can best be illustrated through the example of the "content migration" sub-team. This team acts as a bridge between the editorial function and the fabrication pipelines (which subsequently deliver content for delivery to users of the main product web page). The content is mastered in XML using a variety of DTDs but the definitions in these DTDs are quite broad, in the sense that several different types of content

can be represented using the same DTD. However, for delivery to the production systems certain additional rules must be enforced—these rules (written in Schematron) "tailor" the broad DTD definition for each particular content type.

The Schematron rules effectively represent the editorial requirements for the content—there are many of them and they change quite often. The content migration sub-team develops and maintains these Schematron rules, supplying them to the fabrication team for inclusion in the various pipelines related to those content types. Prior to the development of the Ant library the typical workflow was something like:

1. Find the appropriate build.xml file and edit it, commenting out details of other team members content types and un-commenting their own.
2. Find another, apparently unrelated build.xml file (e.g. with a relative path similar to "../../common/build.xml") and make similar changes.
3. Run the first build file to create a pre-compiled Schematron file (i.e. an XSLT).
4. Manually move the output file to a different folder.
5. Inform the fabrication team that there is a new version of the file.
6. Liaise with fabrication to correct any issues found during the next run of the content through its pipeline. This may include confirming which version of the code is being included and re-compiling if necessary.

Note that as all code was shared in the same TFS repository it was very difficult to go back and check the status of any particular version. Different versions were not explicitly labelled and any changes to the code might be overwritten by another team member committing to the same files in the same repository.

Therefore, after the Ant library was developed a second, higher level set of macros was developed to assist the content migration sub-team. The Schematron files for each content stream were placed in their own Git repository and which had its own (Ant) properties file that described how the Schematron files should be compiled and the various options that should be applied. A standard template for a build.xml file common to all the content streams was also created. The workflow now became:

1. Create a new feature branch in the repository and make the code changes required.
2. From a command line run `$ ant compile` to compile the Schematron files (using the `compile-schematron` macros).
3. Run `$ ant validate` to validate the compiled Schematron against a representative data model, or series of models (using the `validate-schematron` macros).
4. Merge the feature changes back into master (and repeat from step 1 for additional features).
5. Once all features have been finished create a "release" branch,
6. Run the validation tasks once more. If all the tests are valid then run `$ ant version` to update the version number—this version number is now embedded in the compiled Schematron.
7. Prepare release documentation and run `$ ant docs` to prepare it. This documentation will be deployed with the compiled Schematron files so that all content developers can see what changes have been made and why. The build file will check that this documentation has been completed and will not deploy until it has.
8. Run `$ ant deploy` to deploy the compiled Schematron files to the CAPE for reference as well as deploying them directly to a network share where the fabrication pipeline will pick it up automatically. This uses the `deploy-files` macros, but the standard build file will not allow deployment from other than a release branch or if there are outstanding commits. In other words, the developer cannot deploy simply "as a test"—they must think about their actions.
9. Merge the feature branch back into master and push to the server to keep a record of all changes.

This freed up the content migration sub-team to concentrate on the details of the content and the Schematron rules, dealing with actual content issues rather having to continually having to manually configure build files at each step. Furthermore, the addition of a version number meant that any errors encountered in the fabrication pipeline were much easier to track and resolve. Of course, some of these improvements are directly related to the enforcement of the Development Policy—but most of the time this is accomplished *automatically* through the Ant library macros.

4. XPantS

> "A simple library for simple folk."

The success of the Ant library for the content delivery team led the authors to consider developing a clean-room, open source equivalent. This would make similar functionality available to the community at large and allow for more rapid development and bug fixes. Shortly thereafter the XPantS ("XML Practical Ant Scripts) library was developed and has recently been made available on GitHub [https://github.com/encodis/xpants].

There are, of course, a number of differences with the original Ant library as a number of constraints imposed by the original development and production environments no longer apply.

· Unlike the Ant library, XPantS is distributed as a Jar file and referenced in build files via the "antlib" system which brings it inline with similar distributions.
· The macros are also simpler and more flexible and the library includes a basic set of XSLTs in the deployable Jar file.
· Development now uses AntUnit to test the macros themselves and the library now includes a simple wrapper around the XSpec [https://github.com/xspec/xspec] library.
· Documentation for the macros does not require separate hosting. The Markdown is extracted as part of a local build process and uploaded to GitHub Docs.

XPantS has a number of dependencies, including:

· ANT Contrib [http://ant-contrib.sourceforge.net]. XPantS uses some tasks defined by ANT Contrib, particularly the `<for>` and `<propertyregex>` tasks.
· Although it is not required the Saxon XSLT and XQuery Processor [https://www.saxonica.com/products/products.xml] is recommended. XPantS will use Saxon for XSLT processing if it can, otherwise it will default to Xalan.
· XML Resolver [http://www.java2s.com/Code/Jar/x/Downloadxmlresolverjar.htm], installed in Ant's classpath.
· For simplicity the ISO Schematron XSLT files are included directly in the XSLT part of the library.
· The `<convert-schema>` and `<validate-with-schematron>` macros use the Trang [http://www.thaiopensource.com/relaxng/trang.html] converter and Jing [http://www.thaiopensource.com/relaxng/jing.html] validator respectively which should also be available in the Ant classpath.
· A number of macros act as wrappers around command line applications, which obviously need to be installed to work. These include: AWS Command Line Tools [https://aws.amazon.com/cli/], Git [https://github.com/git/git], Pandoc [https://pandoc.org/], Python [https://www.python.org/], SSH [https://www.ssh.com/ssh/], HTML Tidy [http://www.html-tidy.org/] and XSpec [https://github.com/xspec/xspec].

4.1. Macro construction

Most of the macros follow a pattern. Firstly, the main entry point is a macro of the same name as the module (or file). So, for example, the compile-schematron.xml file contains a "top level" `<compile-schematron>` macro. This will detect the type of arguments given to it and run either the `<compile-schematorn-file>` macro if a single file is supplied or the `<compile-schematron-fileset>` macro if a fileset (or folder of files) has been given. Note that in the latter case the "-fileset" macro will call the "-file" macro in a loop over all files in the fileset. This pattern is typical of all macros that can take either a file or a fileset specification.

Macros will also use other macros in the library as required. For example, almost all macros use macros from the attr-checks.xml module to provide additional checks on their arguments, beyond that provided by the Ant `<macrodef>` facility itself.

Although considerable use of the Ant Contrib `<for>` macro is used (simply due to the utility it provides) an attempt has been made to avoid other logic constructs such as the `<if>/<then>/<else>` macros. As Ant is generally declarative in nature rather than imperative, extensive use has been made of its built in "if/unless [https://ant.apache.org/manual/ifunless.html]" attributes.

As an example of macro construction consider the `<xspec-file>` macro below:

```
<macrodef
    name="xspec-file"
    description="Run a single XSpec file">

    <attribute
        name="file"
        default=""
        description="XSpec file"/>

    <attribute
        name="test"
        default=""
        description="Type of test to run [xslt,xquery,schematron]"/>

    <attribute
```

```
            name="coverage"
            default="false"
            description="Output test coverage report [true,false]"/>

        <attribute
            name="junit"
            default="false"
            description="Output JUnit report [true,false]"/>

        <sequential>

            <!-- fail if required attributes not set -->
            <check-attr-set
              macro="xspec-file"
              name="file"
              value="@{file}"/>

            <check-attr-list
              macro="xspec-file"
              name="test"
              value="@{test}"
              list="xslt,xquery,schematron"/>

            <check-attr-bool
              macro="xspec-file"
              name="coverage"
              value="@{coverage}"/>
            <check-attr-bool
              macro="xspec-file"
              name="junit"
              value="@{junit}"/>

            <!-- find XSpec command -->
            <check-exe-path
              property="xspec.exe"
              unix="xspec.sh"
              windows="xspec.bat"
              unless:set="xspec.exe"/>

            <!-- get test type -->
            <local name="test.@{test}"/>
            <property name="test.@{test}" value="true"/>

            <!-- run tidy -->
            <property name="exec.failonerror" value="true"/>

            <exec
                executable="${xspec.exe}"
                failonerror="${exec.failonerror}">
                <arg line="-t" if:set="test.xslt"/>
                <arg line="-q" if:set="test.xquery"/>
                <arg line="-s" if:set="test.schematron"/>
                <arg line="-c" if:true="@{coverage}"/>
                <arg line="-j" if:true="@{junit}"/>
                <arg line="@{file}"/>
            </exec>

        </sequential>
</macrodef>
```

All `<macrodef>` elements and their related `<attribute>`/`<element>` children have "description" attributes—these are used to generate documentation via the extract-markdown.xml module. In the first section, the attributes are checked: the file name cannot be blank, the type of test must be one of "xslt", "xquery" or "schematron" and the values

of the "coverage" and "junit" attributes must be "true" or "false". The various `<check-attr-...>` macros will fail the build if these conditions are not met. (Note that Ant itself will only check if attributes are required or optional, and will apply their default values. It cannot check attribute types or lists of allowed values). Next the system's *path* is checked for the presence of the XSpec executable (this is "xspec.sh" in Unix based systems and "xspec.bat" for Windows). Again the build will fail if the executable cannot be found.

Next a property is set whose name is based on the test type. So, therefore, if the "xquery" test was selected the "test.xquery" property will be created. This is used in the `<exec>` macro to set the correct command line arguments: the "test.xslt" and "test.schematron" properties will not exist and so the "-t" and "-s" arguments will not be included in the call. This pattern is used extensively in the XPantS library to keep the code readable (although it does require familiarity with how Ant properties work).

4.2. XPantS contents

These macros have been developed as they were required, especially the external tool wrappers, but they can be divided into three broad categories.

The XML macros include:

- `apply-transform`: Applies an XSLT (or a series of XSLTs) to a file or fileset. As might be expected this is a wrapper around the built in Ant `<xslt>` task but with some convenience features.
- `compile-schematron`: Compiles a Schematron file (or files) into an XSLT using the ISO Schematron transforms, although the individual stylesheets that perform the various stages of the compilation can be overridden if required. Standard XPantS library XSLTs can also be applied directly, e.g. to remove comments.
- `convert-schema`: A wrapper around the Trang [http://www.thaiopensource.com/relaxng/trang.html] converter (and consequently has the same limitations).
- `validate-instance`: A wrapper around Ant's `<xmlvalidate>` and `<schemavalidate>` tasks, along with a wrapper around the Jing [http://www.thaiopensource.com/relaxng/jing.html] validator for RNG/RNC instance files. This allows the user to use the same macro to validate an instance against any type of schema.

Some of the more utility based macros include:

- `attr-checks`: These macros are mainly used by the rest of the library as part of it's parameter checking and error handling.
- `compare-files`: Compare files (or filesets) using either the *diff* tool or DeltaXML [https://www.deltaxml.com] (which must also be installed).
- `deploy-files`: Deploys a set of files to a target. This may use the standard file system utilities (i.e. the Ant `<copy>` task) but options exist for using the AWS command line tool (for example, deployment to an S3 bucket) or *scp*.
- `extract-markdown`: Uses XSLTs internal to the library to extract Markdown documentation from Ant build files, XSLT files or Schematron files. For Ant build files it also adds documentation around the macros and targets the file contains. XSLT file documentation is enhanced with lists of templates and Schematron files have documentation for pattern, rules, asserts and reports added. The output is Markdown, so may need to be converted into HTML (e.g. using the `<pandoc>` macro, below).
- `file-utils`: These macros are mainly used by the rest of the library as part of it's parameter checking and error handling.

A number of external (command line based) tools also have Ant macro wrappers:

- `aws`: A wrapper around the AWS command line tool. It also includes a macro that uses the "s3 sync" command, which is used by `<deploy-files>`.
- `git`: These macros provide access to the *git* command line functionality. Convenience functions are included for getting the current branch name, getting the current status, checking for outstanding commits (and failing the build if there are any) and checking if the current branch matches a particular pattern (e.g. failing the build if the current branch is not "release).
- `pandoc`: A wrapper for the Pandoc [http://pandoc.org] document converter. Most of the important options are available as arguments to the `<pandoc>` macro.
- `python`: A wrapper for the *python* command to run local scripts or installed modules. Also includes macros to build and install Python Package Index (PyPi) modules.
- `ssh`: Wrappers for the *ssh* (Secure Shell) and *scp* commands. Also used by the `<deploy-files>` macro.
- `tidy`: A wrapper around the HTML Tidy [http://www.html-tidy.org] application.
- `version-number`: One of the newer macros, this contains a number of functions to update version numbers that conform to the Semantic Versioning standard.

- **xspec**: This is simply a wrapper around the xspec.bat/xspec.sh scripts provided by the XSpec install (so it should be on the path).

The AntUnit [https://ant.apache.org/antlibs/antunit/] testing framework is gradually being applied to more of the macros as development time permits. This should increase the stability and quality of the library.

4.3. Build file example

In this example a project's `<deploy>` task first checks that there are no outstanding commits. Then it checks that the current *git* branch matches "release/v."–this may be meeting some team or DevOps requirement. Only then does it deploy the files.

```
<target name="deploy">
    <git-check-outstanding-commits/>
    <git-check-branch pattern="release/v*.*"/>
    <deploy-files target="\\myfile.share\assets">
        <fileset dir="build>
            <include name="*.xml"/>
        </fileset>
    </deploy-files>
</target>
```

5. Conclusion

Although Ant might be considered an old technology by modern standards it can still be employed to good effect. The creation of a standard set of macros has allowed the content delivery team to drastically improve their efficiency, the quality of their code and their interaction with other teams involved in content production by allowing them to concentrate on actual content issues rather than the minutiae of how they build things. It has also positioned the team to be more in line with current CI/CD (Continuous Integration/Continuous Delivery) models that the company is starting to adopt. It is, however, important to note that teams must have a clear idea of how they want to work—in the form of the Development Policy, for example, or a similar document—before factoring this into build files and configurations themselves.

A number of the lessons learned through the development of the team's Ant library are now available in the XPantS library. It is hoped that this will prove useful to the general community.

Bibliography

[1] "Apache Ant", https://ant.apache.org

[2] "Schematron", http://schematron.com

[3] "Semantic Versioning 2.0.0", https://semver.org

[4] "DocBook.org", https://docbook.org

[5] "Markdown: Syntax", https://daringfireball.net/projects/markdown/syntax

[6] "Apache Ivy", http://ant.apache.org/ivy/

[7] "Apache Maven Project", https://maven.apache.org

[8] "XML - Practical Ant Scripts", https://github.com/encodis/xpants

[9] "XSpec", https://github.com/xspec/xspec

[10] "Ant Contrib Tasks", http://ant-contrib.sourceforge.net

[11] "SAXON: The XSLT and XQuery Processor", https://www.saxonica.com/products/products.xml

[12] "XML Resolver", http://www.java2s.com/Code/Jar/x/Downloadxmlresolverjar.htm

[13] "Trang: Multi-format schema converter based on RELAX NG", http://www.thaiopensource.com/relaxng/trang.html

[14] "Jing: A RELAX NG validator in Java", http://www.thaiopensource.com/relaxng/jing.html

[15] "AWS Command Line Interface", https://aws.amazon.com/cli/

[16] "Git", https://github.com/git/git

[17] "Pandoc: a universal document converter", https://pandoc.org/

[18] "Python", https://www.python.org/

[19] "SSH (Secure Shell)", https://www.ssh.com/ssh/

[20] "HTML Tidy", http://www.html-tidy.org/

[21] "DeltaXML", https://www.deltaxml.com"

[22] "Apache AntUnit", https://ant.apache.org/antlibs/antunit/

An XSD 1.1 Schema Validator Written in XSLT 3.0

Michael Kay, Saxonica

Abstract

The paper describes an implementation of an XML Schema validator written entirely in XSLT 3.0. The main benefit of doing this is that the validator becomes available on any platform where XSLT 3.0 is implemented. At the same time, the experiment provides a valuable test of the performance and usability of an XSLT 3.0 implementation, and the experience gained from writing this application can feed back improvements to the XSLT processor that will benefit a large range of applications.

1. Introduction

This paper describes the outline design of an XML Schema (XSD 1.1) validator written entirely in XSLT 3.0.

There were two reasons for undertaking this project:

- Firstly, we see a demand to make implementations of the latest XML standards from W3C available on a wider variety of platforms. The base standards (XML 1.0, XPath 1.0, XSLT 1.0, XSD 1.0) that came out between 1998 and 2001 were very widely implemented, but subsequent refinements of these standards have seen far less vendor support. One can identify a number of reasons for this: perhaps the first versions of the standards met 90% of the requirements, and the market for subsequent enhancements is therefore much smaller; perhaps the early implementors found that their version 1 products were commercially unsuccessful and they therefore found it difficult to make a business case for further investment. In the case of products like libxml/libxslt produced by enthusiasts working in their spare time, perhaps the enthusiasts are more excited by a brand new technology than by version 2 of something well established. Whatever the causes, XSD 1.1 in particular only has three known implementations (Altova, Saxon, and Xerces) compared with dozens of implementations of XSD 1.0, despite the fact that there is a high level of consensus in the user community that the new features are extremely useful.

- Secondly, as editor of the XSLT 3.0 specification and developer of a leading implementation of the language, I felt a need to "get my hands dirty" by developing an application in XSLT 3.0 that would stretch the capabilities of the language. I am often called on to provide advice to XSLT 3.0 users, and even now that the language specification is complete, I find myself designing extensions to meet requirements that go beyond the base standard, and this can only be done on the basis of personal experience in using the language "in anger" for real applications. In addition, the usability of a language compiler depends in high measure on the quality of diagnostics available, and improving the quality of diagnostics depends entirely on a feedback process: you need to make real programming mistakes, spot when the diagnostics could be more helpful, and make the required changes.

More specifically, Saxonica has released an implementation of XSLT 3.0 written in Javascript to run in the browser, and an obvious next step is to extend that implementation to run under Node.js on the server. But for the server environment we need to implement a larger subset of the language, and so the question arose, should we write a version of the schema processor in Javascript? However, Javascript is not the only language of interest: we're also seeing demand for our technology from users whose programming environment is C, C++, C#, PHP, or Python. Since (using a variety of tools) we've been trying to deliver XSLT 3.0 capability in all these environments, a natural next step is to try and deliver a schema processor that will run on any of these platforms, by writing it in portable XSLT 3.0 code.

Architecturally, there are two parts to an XSD schema processor:

- The first part is the schema compiler. This takes as input the source XSD schema documents, and performs as much pre-processing of this input as it can prior to the validation of actual XML instances against the schema. This preprocessing includes verification that the schema documents themselves represent a valid schema, and other tasks such as expanding defaults. The XSD specification from W3C describes a "schema component model" which is an abstract description of a data structure that can be used to represent a compiled schema; and it gives detailed rules for translating source schema documents to this abstract data model. The Saxon schema processor follows this approach very closely, and it also defines a concrete XML-based representation of the schema component model which realises the output of the schema compilation phase in concrete form. This output in fact contains a little more than the SCM components and their properties; it also contains a representation of the finite state machines used to implement the grammar of complex types defined in the schema.

- The second part is the instance validator. The instance validator takes two inputs: the compiled schema (SCM) output by the schema compiler, and an XML instance document to be validated. In principle its output is a PSVI (post schema validation infoset) containing the instance document as a tree, decorated with information derived from schema validation, including information identifying nodes that are found to be invalid. In practice, in the Saxon implementation, there are two outputs: a validation report, which in its canonical form is an XML document listing all the validation errors that were found, and a tree representation of the instance document with added type annotations, in the form prescribed by the XDM data model used by XPath, XQuery, and XSLT. Because the Saxon schema validator is primarily designed to meet the schema processing requirements of XSLT and XQuery, it only delivers the subset of PSVI required in that environment: this means that the output XML tree is only delivered if it is found to be valid, and the only tree decorations added by the validator are references from valid element and attribute nodes to the schema types against which they were validated. In effect, if the input is valid, then the output of the Saxon schema validator is a representation of the input XML instances with added type annotations; while if the input is invalid, then the output is a report listing validation errors.

The software described in this paper is an XSLT 3.0 implementation of the instance validator. It would certainly be possible to write an XSLT 3.0 implementation of the schema compiler, and we may well do that next, but we haven't tackled this yet.

Moreover, the only output of the instance validator described here is a validation report showing the validation errors. The software doesn't attempt to produce an XDM representation of the input document with type annotations. In fact, this isn't possible to do using standard XSLT 3.0 without extensions. XSLT 3.0 has no instructions to output elements and attributes with specific type annotations; the only way it can generate typed output is by putting the output through a schema validator, which is assumed to exist as an external component. So to write that part of the validator in XSLT 3.0, we would need to invent some language extensions.

2. The Validation Task

In this section we describe what the validator actually does.

Let's start with an example of a very simple schema, like this:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="books">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="book" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
  <xs:key name="isbn-key">
   <xs:selector xpath="book"/>
   <xs:field xpath="@isbn"/>
  </xs:key>
 </xs:element>

 <xs:element name="book">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="publisher" type="xs:string"/>
    <xs:element name="author" type="xs:string" minOccurs="1" maxOccurs="5"/>
    <xs:element name="date" type="xs:gYear"/>
    <xs:element name="price" type="moneyType"/>
   </xs:sequence>
   <xs:attribute name="isbn" type="ISBNType" use="required"/>
   <xs:assert test="if (publisher eq 'McGraw-Hill') then starts-with(@isbn, '007') else
                    if (publisher eq 'Academic Press') then starts-with(@isbn, '012')
                    else true()"/>
  </xs:complexType>
 </xs:element>

 <xs:complexType name="moneyType">
  <xs:simpleContent>
   <xs:extension base="xs:decimal">
    <xs:attribute name="currency" type="currencyType"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>

 <xs:simpleType name="currencyType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="USD"/>
   <xs:enumeration value="GBP"/>
   <xs:enumeration value="EUR"/>
   <xs:enumeration value="CAD"/>
  </xs:restriction>
 </xs:simpleType>
```

```
<xs:simpleType name="ISBNType">
 <xs:restriction base="xs:string">
  <xs:pattern value="[0-9]{9}[0-9X]"/>
 </xs:restriction>
</xs:simpleType>

</xs:schema>
```

A valid XML instance conforming to this schema might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book isbn="0070491712">
        <title>Apple PASCAL: a hands-on approach</title>
        <author>Arthur Luehrmann</author>
        <author>Herbert Peckham</author>
        <publisher>McGraw-Hill</publisher>
        <date>1981</date>
        <price currency="USD">13.95</price>
    </book>
    <book isbn="0124119700">
        <title>An Introduction to Direct Access Storage Devices</title>
        <author>Hugh Sierra</author>
        <publisher>Academic Press</publisher>
        <date>1990</date>
        <price currency="USD">72.95</price>
    </book>
</books>
```

The Saxon-EE schema compiler can be invoked to convert the source schema into an SCM file, for example with a command such as:

```
java com.saxonica.Validate xsd:books.xsd -scmout:xsd
```

Here is the resulting SCM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<scm:schema xmlns:scm="http://ns.saxonica.com/schema-component-model"
            generatedAt="2018-06-04T10:39:09.702+01:00"
            xsdVersion="1.1">
   <scm:simpleType id="C0"
                   name="currencyType"
                   base="#string"
                   variety="atomic"
                   primitiveType="#string">
      <scm:enumeration value="EUR"/>
      <scm:enumeration value="CAD"/>
      <scm:enumeration value="USD"/>
      <scm:enumeration value="GBP"/>
   </scm:simpleType>
   <scm:simpleType id="C1"
                   name="ISBNType"
                   base="#string"
                   variety="atomic"
                   primitiveType="#string">
      <scm:pattern value="[0-9]{9}[0-9X]"/>
   </scm:simpleType>
   <scm:complexType id="C2"
                    name="moneyType"
                    base="#decimal"
                    derivationMethod="extension"
                    abstract="false"
                    variety="simple"
```

```
                          simpleType="#decimal">
      <scm:attributeUse required="false" inheritable="false" ref="C3"/>
  </scm:complexType>
  <scm:attribute id="C3"
      name="currency"
      type="C0"
      global="false"
      inheritable="false"
      containingComplexType="C2"/>
  <scm:element id="C4"
             name="book"
             type="C5"
             global="true"
             nillable="false"
             abstract="false"/>
  <scm:complexType id="C5"
      base="#anyType"
      derivationMethod="restriction"
      abstract="false"
      variety="element-only">
      <scm:attributeUse required="true" inheritable="false" ref="C9"/>
      <scm:modelGroupParticle minOccurs="1" maxOccurs="1">
         <scm:sequence>
            <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C10"/>
            <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C11"/>
            <scm:elementParticle minOccurs="1" maxOccurs="5" ref="C12"/>
            <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C13"/>
            <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C14"/>
         </scm:sequence>
      </scm:modelGroupParticle>
      <scm:finiteStateMachine initialState="0">
         <scm:state nr="0">
            <scm:edge term="C10" to="1"/>
         </scm:state>
         <scm:state nr="1">
            <scm:edge term="C11" to="2"/>
         </scm:state>
         <scm:state nr="2">
            <scm:edge term="C12" to="3"/>
         </scm:state>
         <scm:state nr="3" minOccurs="1" maxOccurs="5">
            <scm:edge term="C12" to="3"/>
            <scm:edge term="C13" to="4"/>
         </scm:state>
         <scm:state nr="4">
            <scm:edge term="C14" to="5"/>
         </scm:state>
         <scm:state nr="5" final="true"/>
      </scm:finiteStateMachine>
      <scm:assertion xmlns:xs="http://www.w3.org/2001/XMLSchema"
         test="if (publisher eq 'McGraw Hill') then starts-with(@isbn, '007')
                  else if (publisher eq 'Academic Press') then starts-with(@isbn, '01
                  else true()"
         defaultNamespace=""
         xml:base="file:/Users/mike/Documents/papers/markupuk2018/books.xsd"/>
  </scm:complexType>
  <scm:element id="C6"
             name="books"
             type="C7"
             global="true"
             nillable="false"
             abstract="false">
```

```
      <scm:identityConstraint ref="C8"/>
</scm:element>
<scm:complexType id="C7"
    base="#anyType"
    derivationMethod="restriction"
    abstract="false"
    variety="element-only">
    <scm:elementParticle minOccurs="1" maxOccurs="unbounded" ref="C4"/>
    <scm:finiteStateMachine initialState="0">
       <scm:state nr="0">
          <scm:edge term="C4" to="1"/>
       </scm:state>
       <scm:state nr="1" final="true">
          <scm:edge term="C4" to="2"/>
       </scm:state>
       <scm:state nr="2" final="true">
          <scm:edge term="C4" to="2"/>
       </scm:state>
    </scm:finiteStateMachine>
</scm:complexType>
<scm:key id="C8" name="isbn-key" targetNamespace="">
    <scm:selector xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xpath="book"
                  defaultNamespace=""/>
    <scm:field xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xpath="@isbn"
               defaultNamespace=""
               type="#string"/>
</scm:key>
<scm:attribute id="C9"
    name="isbn"
    type="C1"
    global="false"
    inheritable="false"
    containingComplexType="C5"/>
<scm:element id="C10"
             name="title"
             type="#string"
             global="false"
             containingComplexType="C5"
             nillable="false"
             abstract="false"/>
<scm:element id="C11"
             name="publisher"
             type="#string"
             global="false"
             containingComplexType="C5"
             nillable="false"
             abstract="false"/>
<scm:element id="C12"
             name="author"
             type="#string"
             global="false"
             containingComplexType="C5"
             nillable="false"
             abstract="false"/>
<scm:element id="C13"
    name="date"
    type="#gYear"
    global="false"
    containingComplexType="C5"
    nillable="false"
```

```
              abstract="false"/>
   <scm:element id="C14"
                name="price"
                type="C2"
                global="false"
                containingComplexType="C5"
                nillable="false"
                abstract="false"/>
</scm:schema>
```

Let's look briefly at what this contains. The children of the `scm:schema` element represent different *schema components* such as element declarations, attribute declarations, simple and complex types, each with a unique identifier. For convenience I've rearranged these in order of the component identifier (the actual order doesn't matter).

- C0 is the simple type named `currencyType`. It is an atomic type derived from `xs:string` (the built-in type is represented as `#string`). The `scm:enumeration` elements list the permitted values.

- C1 is the simple type named `ISBNType`. It is an atomic type derived from `xs:string`, with a pattern facet constraining the permitted values.

- C2 is the complex type named `moneyType`. It is a "complex type with simple content", allowing simple content of type `xs:decimal`, and an attribute. The complex type contains an `scm:attributeUse` element which is a reference to the attribute declaration defining the attribute; like all references from one component to another, this uses the component identifier, in this case `C3`.

- C3 is the attribute declaration for `currency`; it is a local declaration (`global="false"`). The type of the attribute is defined by a reference to the simple type component C0.

- C4 is the element declaration for the `book` element; it is defined largely by a reference to the complex type C5 which defines the allowed content.

- C5 is the complex type defining the allowed content of `book` elements. The complex type itself has no name. The type is derived by restriction from `xs:anyType`, and the permitted content is defined in terms of a `modelGroupParticle` containing a sequence of `elementParticles` representing the permitted child elements: these are references to local element declarations appearing later in the SCM file. The complex type component also contains a representation of the (deterministic) finite state machine used to check instances against the grammar defined in the source schema. This defines a set of states (the initial state is 0, the final state is 5) and the permitted transitions between them. The transitions (edges) are defined by reference to the schema components for the contained element particles. Finally, the complex type component contains the XPath assertion that constrains the relationship between publishers and ISBNs. The `scm:assertion` element includes an `xml:base` attribute because it is possible (in theory) for the evaluation of the XPath assertion to depend on the base URI of the containing element in the source schema.

- C6 is the element declaration for the outermost `books` element: it refers to the complex type definition C7 and the identity constraint (the `xs:key` constraint) C8.

- C7 is the complex type definition for the outermost `books` element. Like C5, it contains a (very simple) finite state machine used to enforce the grammar.

- C8 represents the `xs:key` constraint specifying that ISBNs must be unique.

- C9 to C13 are the attribute and element declarations for the details of a book, and are all very simple. The meanings of the attributes are very closely aligned with the properties of the abstract Schema Component Model defined in the W3C specification.

Given this schema and this instance document, the task of the validator is to produce an empty validation report showing that there are no errors. The validation report becomes more interesting if the instance is invalid. For example, we can use this command:

```
java com.saxonica.Validate -xsd:books.scm -s:books-invalid.xml -report:report.xml
```

to validate this invalid instance:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<books>
    <book isbn="0070491712">
        <title>Apple PASCAL: a hands-on approach</title>
        <publisher>McGraw-Hill</publisher>
        <date>1981</date>
        <price currency="NZD">13.95</price>
    </book>
    <book isbn="0134119700">
        <title>An Introduction to Direct Access Storage Devices</title>
        <author>Hugh Sierra</author>
        <publisher>Academic Press</publisher>
        <date>1990-04</date>
        <price currency="USD">72.95</price>
    </book>
</books>
```

and the result is the following report:

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-report xmlns="http://saxon.sf.net/ns/validation"
                   system-id="file:/Users/mike/Documents/papers/markupuk2018/
                   books-invalid.xml">
    <error line="6"
           column="15"
           path="/Q{}books[1]/Q{}book[1]/Q{}date[1]"
           xsd-part="1"
           constraint="cvc-complex-type.2.4">In content of element &lt;book&gt;: The
             content model does not allow element &lt;Q{}date&gt; to appear immediately
             after element &lt;publisher&gt;. No further elements are allowed at
             this point. </error>
    <error line="7"
           column="31"
           path="/Q{}books[1]/Q{}book[1]/Q{}price[1]/@currency"
           xsd-part="2"
           constraint="cvc-complex-type.3">Value "NZD" contravenes the enumeration
           facet "EUR, USD, CAD, GBP" of the type Q{}currencyType</error>
    <error line="11"
           column="17"
           path="/Q{}books[1]/Q{}book[2]/Q{}author[1]"
           xsd-part="1"
           constraint="cvc-complex-type.2.4">In content of element &lt;book&gt;:
             The content model does not allow element &lt;Q{}author&gt; to appear
             immediately after element &lt;title&gt;. No further elements are allowed
             at this point. </error>
    <error line="13"
           column="15"
           path="/Q{}books[1]/Q{}book[2]/Q{}date[1]"
           xsd-part="2"
           constraint="cvc-datatype-valid.1">The content "1990-04" of element &lt;date&gt
             does not match the required simple type. Cannot convert '1990-04' to a
             gYear</error>
    <error line="9"
           column="29"
           path="/Q{}books[1]/Q{}book[2]"
           xsd-part="1"
           constraint="sec-cvc-assertion.0">Element book does not satisfy assertion
             if (publisher eq 'McGraw Hill') then starts-with(@isbn, '007') else
             if (publisher eq 'Academic Press') then starts-with(@isbn, '012')
             else true()</error>
    <meta-data>
        <validator name="SAXON-EE" version="9.9.0.1"/>
        <results errors="5" warnings="0"/>
        <schema file="books.scm" xsd-version="1.1"/>
```

```
        <run at="2018-06-04T11:12:24.651+01:00"/>
    </meta-data>
</validation-report>
```

The report shown here comes from the existing Saxon-EE validator written in Java. Our task is to reproduce this report with a validator written entirely in portable XSLT.

3. Design Considerations

This section describes some of the design challenges posed.

3.1. Generic Stylesheet or Generated Stylesheet?

At the beginning of this exercise we considered two alternative designs: the validator could run as a generic XSLT stylesheet taking its rules dynamically from the SCM input document, or it could be a custom XSLT stylesheet generated from the SCM input document and dedicated to doing validation against this particular schema.

Both approaches have potential advantages, but we chose the first on the grounds of simplicity. As we will see later, there are some technical challenges where the second approach might have afforded a solution.

3.2. Subset of XSLT 3.0

We decided that, if possible, the validator should be written in the subset of XSLT 3.0 that is supported by Saxon-JS (including the use of `xsl:evaluate` which requires a later release of Saxon-JS). This decision means that we cannot use higher-order functions: these are an optional XSLT 3.0 feature which Saxon-JS does not currently support.

3.3. Streaming

In an ideal world, we would use an XSLT 3.0 streaming transformation to process the input instance document, so that it does not need to be held completely in memory.

The existing Saxon validator, written in Java, uses streamed processing whereever possible. The main case where streaming is not possible is in evaluating XSD 1.1 assertions: assertions can use arbitrary XPath expressions to process the subtree of the source document rooted at the element to which the assertion applies. The existing validator starts building an in-memory tree when it encounters such an assertion; in the absence of such assertions it is full streamed. It would be possible in principle to avoid building the subtree if the assertion uses a streamable subset of XPath, but the validator does not attempt this.

Emulating this behaviour in the new XSLT validator might be possible, but it is not easy, and in the current project we have not attempted it. One of the main reasons for this is that there are other XSD features (notably the evaluation of uniqueness and referential constraints) for which a streamed implementation is even more difficult.

The main constraint here is that `xsl:evaluate` (the XSLT 3.0 instruction to perform evaluation of a dynamically-constructed XPath expression) is not streamable, because static analysis has no access to the XPath expression in question, and streamability analysis is always done statically. Since `xsl:evaluate` is essential to enable XSD 1.1 features such as assertions and type alternatives to be evaluated, this is a stopper. We might be able to get around it by using the alternative design considered (a generated stylesheet in which the XPath expressions become statically analyzable), but we decided not to go that way.

3.4. Typed data

We have already mentioned that XSLT 3.0 can only generate a typed result tree (specifically, a node with non-trivial type annotations) by invoking a schema validator to produce the type annotations, and this precludes the possibility of writing that schema validator in XSLT 3.0. So the first obvious restriction we have to live with is that our validator will only do that part of the job concerned with detecting invalidity, not the other part concerned with augmenting the supplied instance with type information.

Unfortunately, even the task of distinguishing valid from invalid documents requires some use of type information associated with nodes. The most obvious example is that assertions (XPath expressions in `xsd:assert` declarations) are defined to operate on "semi-typed" data - that is data that has been validated using all the constraints in the schema other than assertions, and typed accordingly. For example, if `@price` and `@discount` are attributes of type `xs:decimal`, then the assertion `test="@discount lt @price"` is defined to do a decimal comparison, not a string comparison.

We don't currently have a solution to this problem. We found, however, that very few schemas are affected. With such schemas, it is currently necessary to rewrite the assertion to do explicit typing: in this case `test="xs:decimal(@discount) lt xs:decimal(@price)"`

Another case where typed data can be important is in `key` and `keyref` constraints. For example, if a uniqueness constraint applies to an attribute `@birthDate` of type `xs:date`, then values of that attribute are compared as dates, not as strings. Again, we found that this is rarely a problem in practice, but in this case we do have a solution that covers most cases: by doing static analysis of the XPath expressions used in the `selector` and `field` elements of the constraint, we can usually determine the data type of the values these expressions are selecting, and we can put this inferred type into the SCM and use it to cast the values before comparison. The only case where this approach proves inadequate are pathological cases where the type cannot be statically inferred, for example when the XPath expressions use union types or wildcards.

3.5. Support for xsi:schemaLocation

The current project is implementing a validator only, with no access to a schema compiler.

The `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes allow an instance document to reference the location of a (source) schema document containing a schema for the instance document. Because this validator has no access to a schema compiler, it cannot implement this capability.

This isn't a fundamental show-stopper. Support for these attributes isn't required for XSD conformance, and many applications avoid using them (there are a number of arguments against using them, not least that when you validate a document, it's because you don't trust it to be valid, and if you don't trust it to be valid, why should you trust it to tell you where the schema is?)

If we had a schema compiler written in XSLT, then it would of course be possible to interface this.

4. Implementation

This section of the paper is not a complete top-down documentation of the working implementation. Rather it gives a selection of snapshots into interesting aspects of the implementation, showing some of the techniques used and obstacles encountered. It is particularly focused on showing how new features in XSLT 3.0 proved valuable.

4.1. Use of Maps for Returned Values

A classic problem with functional programming is that a function can only return one result. If you want to compute two values (say a maximum and minimum) from the same input then you have two choices. You can either process the input more than once (which may involve some redundant computation), or you can return a composite result.

In this application there are many cases where we need to return a composite result.

To take an example, suppose we are validating an attribute, and we find that the declared type for that attribute is a user-defined list type, where the item type of the list is `part-number`, and `part-number` is derived from `xs:ID`. The calling code wants to know (a) whether the attribute is valid against its declared type; (b) what error messages to report if not; and (c) whether the value contains any `xs:ID` or `xs:IDREF` values that need to be added to global tables of `xs:ID` and `xs:IDREF` values for document-level validity checking at the end.

Our solution to this is that we recurse through the instance document in a tree-walk driven by `xsl:apply-templates` in the normal way, but the return value from `xsl:apply-templates` is a map containing all the information gleaned from the processing of this subtree.

Very often the information returned from several calls on `xsl:apply-templates` (for example, one call for child elements and another for attributes) will need to be combined into a single map. At the top level, when we return from the initial call on `xsl:apply-templates` on the root node, all the information that is needed to produce the validation report is present in one large map, and the final stage of processing takes this map and generates the XML report.

The maps that are produced by the different processing stages thus typically include some subset of a common set of fields. These include:

Table 1. The structure of maps used to return partial results of processing

Name	Value
valid	An `xs:boolean` indicating whether the subtree is valid

Name	Value
errors	A set of error objects indicating error information to be included in the validation report
value	The typed value of an element or attribute
type	The type against which a subtree was validated
lexical	The lexical form of an attribute or text node after whitespace normalization
id	A set of `xs:ID` values found in the subtree
id-map	A mapping from `xs:ID` values found in the subtree, to the nodes on which they appeared
idrefs	A set of `xs:IDREF` values found in the subtree

When two of these maps representing properties of different subtrees are combined, different rules apply to each field. For example, for the `id` and `idrefs` and `errors` fields we can take the union of the two values. For the `valid` property, we can apply a logical AND; a tree is valid only if all its subtrees are valid. For `value` and `type` we can drop the value; these fields are used only at the next level up, and do not propagate all the way to the root.

4.2. Declaring Map Types

Standard XSLT 3.0 allows maps (as described in the previous section) to be returned from templates, stored in variables, and so on, so this style of processing is perfectly possible without departing from the standard. However, the facilities for declaring the type of these maps are very weak. The closest we can get is `map(xs:string, item()*)` which is satisfied by any map whose keys are strings.

Much of the debugging process for this stylesheet involves understanding the contents of these returned maps, and it is therefore frustrating that the XSLT 3.0 type system is so poor at validating these maps and reporting type errors. Saxon therefore introduces an extension to XSLT 3.0, called tuple types. Here is a declaration of the returned structure as a tuple type:

```
tuple( valid: xs:boolean?,          (: indicates whether the subtree is valid
                                        (default = true) :)
       errors: element(vr:error)*,  (: a list of errors found when validating the
                                        subtree :)
       value: xs:anyAtomicType*,     (: the typed value of an element or attribute :)
       type: xs:string*,             (: the types of the typed values, as component
                                        IDs :)
       lexical: xs:string?,          (: the lexical form of a value after whitespace
                                        normalization :)
       id: xs:string*,               (: any ID values found while validating an
                                        element or attribute :)
       id-map: map(xs:string, element())*,  (: a mapping from ID values to elements :)
       idrefs: xs:string*)           (: any IDREF values found while validating an
                                        element or attribute :)
```

This clearly documents the expected contents of the map much more precisely than the bland declaration `map(xs:string, item()*)`.

Tuples in Saxon are not a separate data type in the way that maps and arrays are separate data types. Rather, a tuple type is an alternative way of constraining the content of a map. It defines the (string-valued) keys that can appear in the map, and for each permitted key, the permitted type of the corresponding values. Declaring the expected type of a map in this form gives much improved static and dynamic type checking. For example, attempting to reference a non-existing field using the lookup expression `$result?Value` can generate a static error message, as can its use in an inappropriate context such as `$result?valid eq "true"`.

Because tuple type declarations are often quite lengthy, as in this example, Saxon allows them to be declared once using a type alias:

```
<saxon:type-alias name="validation-outcome" type="
  tuple( valid: xs:boolean?,          (: indicates whether the subtree is valid
                                          (default = true) :)
         errors: element(vr:error)*,  (: a list of errors found when validating the
```

```
                                                   subtree :)
            value: xs:anyAtomicType*,    (: the typed value of an element or attribute :)
            type: xs:string*,            (: the types of the typed values, as component
                                             IDs :)
            lexical: xs:string?,         (: the lexical form of a value after whitespace
                                             normalization :)
            id: xs:string*,              (: any ID values found while validating an
                                             element or attribute :)
            id-map: map(xs:string, element())*,  (: a mapping from ID values to elements :)
            idrefs: xs:string*)          (: any IDREF values found while validating an
                                             element or attribute :)

  "/>
```

And the type can then be referenced wherever an as attribute can appear, for example:

```
<xsl:template match="*" as="map(xs:string, item()*)" saxon:as="~validation-outcome"/>
```

The syntax of saxon:as is an XPath SequenceType augmented with Saxon-specific syntax, in this case a reference to a type alias marked as such by the presence of the leading tilde (~). The semantics of saxon:as are that it provides type information additional to that contained in the as attribute. Because (under the XSLT extensibility rules) attributes in the Saxon namespace are ignored by XSLT processors other than Saxon, this whole mechanism enables Saxon to do extra compile-time and run-time type checking, without in any way sacrificing the interoperability of the stylesheet: it still functions correctly under other standards-conforming XSLT 3.0 processors.

4.3. Assessment against Complex Types using Finite State Machines

As we've seen, the finite state machines used to evaluate a sequence of elements against the grammar rules for a complex type are constructed by the schema compiler and embedded in the SCM file that is used as input to the validator.

A simplified validator for a simple finite state machine could be written like this:

```
<xsl:iterate select="$node/*">
    <xsl:param name="state" select="$initial-state" as="element(scm:state)"/>
    <xsl:on-completion>
        <xsl:if test="not($state/@final = 'true')">
            <xsl:sequence select="map{'errors':
                                        scm:error($node, 'Element content is
                                        incomplete')}"/>
        </xsl:if>
    </xsl:on-completion>
    <xsl:variable name="matching-edge" as="element(scm:edge)?"
        select="$state/scm:edge[scm:get(@term)[@name = local-name(current())
                and string(@targetNamespace) = namespace-uri(current())]]"/>
    <xsl:variable name="matching-wildcard-edge" as="element(scm:edge)?"
        select="$state/scm:edge[scm:get(@term)[self::scm:wildcard[
                                    scm:wildcard-matches($containing-type, .,
                                    current())]]]"/>
    <xsl:choose>
        <xsl:when test="empty($matching-edge) and empty($matching-wildcard-edge)">
            <xsl:break select="map{'errors': scm:error(., 'Element ' || name()
                                            || ' is not allowed here')}"/>
        </xsl:when>
        <xsl:when test="empty($matching-edge)">
            <xsl:variable name="wildcard"
                        select="scm:get($matching-wildcard-edge/@term)"
                        as="element(scm:wildcard)?"/>
            <xsl:sequence select="scm:check-wildcard-match($containing-type,
                                    $wildcard, .)"/>
            <xsl:next-iteration>
                <xsl:with-param name="state"
                                select="$states[@nr =
```

```
                                                      $matching-wildcard-edge/@to]"/>
                </xsl:next-iteration>
            </xsl:when>
            <xsl:otherwise>
                <xsl:variable name="decl"
                              select="scm:get($matching-edge/@term)"
                              as="element(scm:element)"/>
                <xsl:apply-templates select="." mode="explicit-decl">
                    <xsl:with-param name="decl" select="$decl"/>
                </xsl:apply-templates>
                <xsl:next-iteration>
                    <xsl:with-param name="state"
                                    select="$states[@nr =
                                        $matching-edge/@to]"/>
                </xsl:next-iteration>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:iterate>
```

The way this code works is as follows:

- The `xsl:iterate` instruction is new in XSLT 3.0. It is rather like `xsl:for-each`, except that it processes the selected items strictly in sequence; the code for processing one item can set parameters for processing the next item; and it is possible to break out of the loop early. The same effect could be achieved with a recursive template, but `xsl:iterate` is often easier to understand. In this case we are iterating over the children of the element being validated.

- There is a single parameter, the current state, which is initially set (by the calling code) to the state numbered 0.

- The `xsl:on-completion` instruction is executed when we reach the end of the sequence of child elements. If the current state is a final state, we return nothing (meaning all is well, the input is valid). Otherwise we return a map containing an error value.

- There are two kinds of transition possible in a given state: named element transitions, and wildcard transitions. We first find all the matching named element transitions (the schema compiler will have ensured there can be at most one) and all the matching wildcard transitions.

- If both sets are empty, there is no legal transition for the current child element in this state, so we return an error value.

- If there is a wildcard transition possible, but no named-element transition, then we check that the wildcard transition is really allowed and that the element is valid against the wildcard (this will take account of its `processContents` attribute, and then proceed to process the next child element in the state reached by this transition.

- If there is a named-element transition possible, then we call `apply-templates` to check that the child element is valid against the required type for the named element, and then proceed to process the next child element in the state reached by this transition.

The actual logic is more complex than this. Firstly, we use a finite state machine with counters, to reduce the size of the finite state machine needed for a grammar such as `<element name="book" minOccurs="100" maxOccurs="200"/>`. Secondly, XSD 1.1 allows "open content" which allows elements matching a given wildcard to appear either (a) anywhere (interleaved content), or (b) at the end of the sequence (suffix content). The possibility of open content is not integrated into the finite state machine, but is instead handled by the validator as it arises. However, the basic principle is retained of stepping through the children using `xsl:iterate` to maintain the current state.

4.4. Checking Assertions

The code for checking input data against assertions defined in the schema is very straightforward. Here is the actual logic (no simplifications this time):

```
<xsl:function name="scm:check-assertions" as="map(*)" saxon:as="tuple(error:
        element(vr:error)*)">
    <xsl:param name="type" as="element(scm:complexType)"/>
    <xsl:param name="node" as="element()"/>
    <xsl:variable name="copy-sans-comments" as="element()">
```

```
            <xsl:apply-templates select="$node" mode="copy-sans-comments"/>
        </xsl:variable>
        <xsl:variable name="failures" as="element(vr:error)*">
            <xsl:for-each select="$type/scm:assertion">
                <xsl:try>
                    <xsl:variable name="assertion-result" as="item()*">
                        <xsl:evaluate xpath="@test"
                            context-item="$copy-sans-comments"
                            namespace-context="scm:make-namespace-context(.)"
                            base-uri="{base-uri($scm)}">
                            <xsl:with-param name="value" select="$copy-sans-comments"/>
                        </xsl:evaluate>
                    </xsl:variable>
                    <xsl:if test="not($assertion-result)">
                        <xsl:sequence select="scm:error($node, ' must satisfy assertion '
                        || @test)"/>
                    </xsl:if>
                    <xsl:catch errors="*">
                        <xsl:sequence select="scm:error($node, ' must satisfy assertion '
                        || @test || '.
                                Evaluation of the assertion failed with a dynamic error: '
                                || $err:description)"/>
                    </xsl:catch>
                </xsl:try>
            </xsl:for-each>
        </xsl:variable>
        <xsl:sequence select="map{'errors': $failures}"/>
</xsl:function>
```

Notes relating to this code:

- The function returns a map, but the `saxon:as` declaration reveals that there is only one field defined in this map, namely the `errors` field. If the constraint is satisfied, an empty map is returned. The reason for defining it this way is that the calling code can use its standard mechanism for combining the results of different validation processes.

- The function makes a copy of the element being validated, in which comments and processing instructions have been removed. This is prescribed by the specification. A copy of the subtree is needed to ensure that the XPath expressions in the assertion have no access to nodes in the input document that fall outside the subtree being validated.

- The assertion is evaluated using `xsl:evaluate`, a new XSLT 3.0 instruction that evaluates XPath expressions known only dynamically (in this case, an expression read from the SCM file). The instruction provides machinery to establish the static and dynamic context for evaluating the expression, here including the context item, the value of the `$value` variable, the namespace context, and the base URI.

- If the effective boolean value of the assertion result is false, the function returns an error value.

- If a dynamic error occurs while evaluating the assertion, this is caught using the new `xsl:try` instruction in XSLT 3.0, and the function returns an error value.

- The whole process is repeated for each defined assertion. If more than one assertion fails, then more than one error will be returned.

4.5. Other Complications

It's worth mentioning a few other complications that the implementation has to deal with, without going into gory detail:

- Gregorian types. XSD 1.1 introduces a new facet which allows you to specify that the timezone on a date/time value is mandatory or optional. It turns out to be easy to check this using XPath expressions for an `xs:date`, `xs:time`, or `xs:dateTime`; but there's no easy way to do it for `xs:gYear`, `xs:gYearMonth`, and friends. Similarly, XSD 1.1 allows facets to control the range of these values, for which XPath offers no support. The validator therefore includes a library of functions for handling the Gregorian types.

- Regular Expressions. The syntax for regular expressions contained in the XSD pattern facet is very similar to the syntax for the XPath `fn:matches()` function -- but not quite the same. Most of the differences are extensions in the

XPath version (for example, support for back-references), and since the schema compilers has done static validation on the expression, we can ignore these differences. The remaining difficulty is the characters "^" and "$", which represent themselves in XSD, but are meta-characters in XPath. To handle this we need to pre-process the regular expression to replace occurrences of "^" and "$"(if not within square brackets) by "\^" and "\$".

- Equality semantics. The equality matching rules in XSD 1.1, used for example by the enumeration facet or in key/keyref comparison, don't correspond directly to any of the ways of testing equality in XPath. For example, in XPath the integer 3 and the double 3e0 are equal, in XSD they are not. This also makes it difficult to use XPath maps to enforce uniqueness constraints. It is therefore necessary to use a custom function for comparing atomic values, and a function `schemaComparable(x)` with the property that `schemaComparable(x)  eq schemaComparable(y)` under the XPath rules if and only if `x` and `y` are equal under the XSD rules.

5. Results

This section attempts to assess the quality metrics of the completed validator.

Saxonica's quality objectives for all its software are conformance, usability, and performance, in that order. We therefore assess the validator against these criteria.

5.1. Conformance

W3C publishes a comprehensive test suite for XSD 1.1. The XSLT validator is currently passing 41330 out of 41363 tests. This is after excluding tests that are not applicable, for example, tests that rely on `xsi:schemaLocation`. This level of conformance comfortably exceeds that of many widely-used schema processors, and the failures largely involve edge cases that few users will ever encounter.

5.2. Usability

Usability is measured primarily by the quality of the error messages. This is not yet as good as the Java validator. Here is the validation report produced for the invalid booklist document supplied earlier:

```
<vr:validation-report xmlns:vr="http://saxon.sf.net/ns/validation">
    <vr:error path="/Q{}books[1]/Q{}book[1]/Q{}date[1]">Element date is not allowed
    here</vr:error>
    <vr:error path="/Q{}books[1]/Q{}book[2]/Q{}author[1]">Element author is not
    allowed here</vr:error>
    <vr:error path="/Q{}books[1]/Q{}book[2]"> must satisfy assertion
                            if (publisher eq 'McGraw Hill')
                            then starts-with(@isbn, '007')
                            else if (publisher eq 'Academic Press')
                            then starts-with(@isbn, '012') else true()</vr:error>
</vr:validation-report>
```

In comparison with the report produced by the Java validator, we see:

- There are no line and column numbers associated with the errors, only a path. This omission is because XSLT provides no standard way of obtaining the line or column number of a node. Some versions of Saxon provide extension functions to get this information, but we want to avoid using extensions; and in any case, a key target environment is Saxon-JS, where we rely on the Javascript DOM as our data model, and the Javascript DOM does not provide line and column information.

- There is no information about which constraint in the XSD specification is violated (this information is of little value to most users, but it is required by the conformance rules).

- There are only three errors reported, rather than five. This is because the XSLT validator is quicker to stop validating sibling elements once one of them has been found to be in error.

So there is some work still to be done to get the usability up to the level of the existing validator.

5.3. Performance

No serious work on optimizing (or measuring) performance has yet been done. However, it's useful to get some very preliminary data to assess whether performance is going to be a major obstacle to the feasibility of the approach.

I constructed a valid data file containing ten thousand book elements.

With the existing Saxon-EE schema validator, validation took 1.1 seconds.

With the XSLT validator, validation (using Saxon-EE on Java as the XSLT engine) took 9.4 seconds.

This represents a ballpark estimate of the relative efficiency. It's not a thorough benchmark in any way; there is no point in doing a thorough benchmark until some basic performance tuning has been done.

There are clearly many opportunities for performance improvement. Some of the obvious inefficiencies, relative to the Java validator, include:

· In evaluating items with a pattern facet, the regular expression is recompiled every time an item is validated. This is because the `fn:matches()` function precompiles the regular expression if it is known statically, but it makes no attempt to cache the compiled regular expression if the same regex is used repeatedly. The regex in this case is read from the SCM file at run-time, so no compile-time optimization is possible.

· Similarly, XPath expressions used in assertions may be recompiled every time they are used. There are some circumstances in which `xsl:evaluate` will cache compiled XPath expressions that are used repeatedly, but this doesn't appear to be happening in this stylesheet.

· Too much data is being retained from validation and passed upwards from the validation of a child to the validation of its parent. This results in bloated maps containing validation outcomes, that take a long time to combine. It's probably not difficult to find "low-hanging" optimizations in this area.

It's clear that a lot of the time is being spent creating and combining the maps that are used to pass data up the tree. The whole application relies very heavily on maps, and its performance depends on the performance of map operations such as `map:put` and `map:merge`. It's possible that it might benefit from a different implementation of maps that is tailored to the usage patterns that occur when map types are declared as tuple types. It could also benefit from changes to the application to make more selective use of maps. In particular, we seem to be incurring heavy costs inspecting and copying maps that are actually empty, because all the data is valid: there's clearly an opportunity for optimizations here.

For the moment, all we can conclude about performance is that more work needs to be done. Making the XSLT validator as fast as the existing Java validator is probably unachievable, but we should be able to get acceptably close.

6. Conclusions

Firstly, we have shown that writing an XSD 1.1 validator in XSLT 3.0 is feasible, with a few caveats: the main limitation is that XSLT 3.0 does not allow the creation of typed element and attribute nodes except by use of an XSD validator, which creates a recursive dependency. This limitation could probably be solved by means of a few simple XSLT extension functions.

Whether an XSD 1.1 validator written in this way can achieve an acceptable level of usability and performance remains an open question. We're certainly close, but we're not quite there yet.

The experiment has certainly yielded insights on how to design complex applications to take advantage of new XSLT 3.0 capabilities; and it has provided usability feedback that has led directly to improvements in the XSLT 3.0 processor, for example in the way type-checking errors with maps are reported and with features for tracing and diagnostics.

It has not so far proved possible to write a streaming validator by exploiting the streaming capabilities of XSLT 3.0. The main obstacle is that the `xsl:evaluate` instruction is intrinsically unstreamable because it cannot be statically analyzed. One way around this problem would be to generate a custome stylesheet to perform validation against a specific schema. Another approach might be to allow a stylesheet author to assert that an `xsl:evaluate` instruction is streamable, and to have the XPath compiler check this assertion when the instruction is evaluated.

When Overlapping XML Meets Changing XML Does Confusion Reign?

Robin La Fontaine

Abstract

The issue of how best to represent overlapping hierarchy in XML has been the topic of a number of papers over the years. This paper is a further contribution to this important issue, but approaching the problem from a different direction. Our goal is to represent changes to documents, and one type of change is change to the markup hierarchy. Therefore our ultimate goal is to be able to represent not only changes to the hierarchy, typically resulting in overlapping hierarchy, but also changes to attributes and text. This is a more ambitious goal than simply representing overlapping hierarchy, and one aspect of this is to make a clear distinction between the different hierarchical structures and the text that corresponds with each one.

Our work started with a delta format for two or more documents, which easily represents inline changes, but handles hierarchy change by duplicating content. In order to avoid duplication, we introduce a distinction between the name of the element (its tag) and the element content, so that assertions can be made separately. We then introduce @dx (change) and @dxTag (change tag) attributes to mark changes. This representation allows us to define overlapping hierarchies in a completely XML way without declaring a dominant hierarchy and while keeping element fragmentation to a minimum. While this solution probably will not scale for large numbers of variants, it shows promise for many classes of documents.

1. Introduction and Background

Our focus has been on the representation of change to structured documents, and in this work our original objective was to find a way to represent change to structure, and this turned out to provide a useful representation for overlapping hierarchy, but with the advantage that other changes could also be represented.

Jeni Tennison says in one of her excellent blogs, "Overlap is arguably the main remaining problem area for markup technologists." [1 [162]]. She points out that this is not only an issue for academics looking at poetry and historical documents, but is also an issue in managing change to structured documents. The example she cites is legislation which is amended over time where the authors are not concerned about changes to structure, their primary interest is in the textual changes.

There are a number of different approaches to this problem, and some excellent reviews of the advantages and disadvantages of the approaches [2 [162]] [3 [162]] [4 [162]] [5 [162]]. Our own goal is to represent changes to documents, such as versions of documents over a period of time as they are amended, and to represent them in a way that is easy to process. This reflects the classic advantage of XML, where content can be re-purposed to meet different needs. If the document can be re-purposed, then we need to be able to re-purpose changes also, and this means changes need to be represented in way that is easy to process.

Ignoring for the moment changes to attributes, most changes can be represented by the addition and deletion of elements and their content. Additionally, we need to be able to mark segments of text that are either added or deleted. This approach allows us to represent any change, although not always in an optimal way. For example, in the extreme, the deletion of the 'old' document and addition of the 'new' document correctly represents the changes, but not in a very useful way. This leads to the observation that by duplicating content it is always possible to represent a change in a structured document. The problem is that we do not wish to duplicate content because this appears to the user as a change to the content, whereas in practice the only change may be to the structural markup around that content. This leads to the need to represent the addition, deletion, and overlapping of structural elements representing hierarchy.

The TEI format [6 [162]] has powerful, though complex, ways of representing different hierarchies, and also variants of text within a document. The goal is to provide rich semantic information about the document, representing all of this information in a single place. Using this semantically rich representation, it would be possible to generate all the different variants of the document, including variants of the text and variants of the hierarchy. When we are considering change, it is essentially all these different variants that we use as a starting point. Therfore in this respect our goal is very similar to, but not quite the same as, the goal of the TEI format. As our starting point is a set of document variants, it is natural that we clearly identify each of these source variants in the single merged document. We therefore always make a very precise differentiation between two overlapping structures, because these are considered to have come from different source documents.

The inherent model that we adopt here, i.e. one that addresses the representation of variants of the whole document, is important because it does differ from a model where the desire is to represent variants in structure within a document. The latter model can lead to a very large number of whole document variants, and our model is not well suited to a large number of variants because the attribute values representing the variants become long and therefore difficult to manage. Our model addresses primarily overlap in the context of change to a document and is not intended as a solution to all overlap representation problems.

Although TEI has these mechanisms, most XML document formats, such as DITA[7 [162]] or DocBook[8 [162]], do not and would therefore benefit from a way of representing overlap. In these formats, overlap representation is needed in order to better represent change. There is a clear advantage to having a standard way to enhance an existing schema with change and overlap representation because structured document editing applications then need to understand only one way of handling this. Schmidt [9 [162]] suggests that a good way to manage documents that have overlapping hierarchy is to split them into separate documents and merge them as needed, though this idea does not seem to have gained a significant following.

There is another distinguishing feature of this solution. In other solutions for representing overlap, identifier attributes (which may or may not be strictly of type xml:id) are often used to indicate which fragments are part of the same element, but with this solution there is no such use of identifier attributes. The problem with using identifier attribtues is that it is difficult to denote a fragment that is part of two separate hierarchies because only one identifier attribute can be present on each element. The identifier attribute could contain a list of identifiers but this does lead to make it more difficult to process.

The representation described here is pure XML. As such, standard XML processing tools such as XSLT and XQuery can be used to process it. Each of the original document variants can be extracted: this was our primary goal and is an

important feature. We have verified that it is quite simple in XSLT to extract a single version, and it is simple to determine the ancestors of a particular element or piece of text. We are currently researching alternative types of processing. One XSLT approach shows particular promise for processing n-way comparison results. This uses a template that employs sibling recursion and XSLT 3.0 maps, the maps keep track of the state of each tree using an extension to the principle of a common stack.

There are validation rules, which we express in Schematron, for this representation. Validation against the original schema of the source documents would need to be done by extracting each version and validating it. In other words, we can assert that the representation is correct if the Schematron rules are passed and if we can extract each of the original documents correctly, i.e. the extracted document is deep equal to the original.

2. How Content Duplication Represents Any Change

Our starting point was an existing solution (a delta format) for representing change to elements, attributes and text in XML documents.[1] Any change could be represented, but changes to structure required some duplication of content. For example, two paragraphs (denoted A and B) might be:

```
<p>The quick brown fox.</p>
```

and

```
<p>The <s>quick</s> brown fox.</p>
```

This is a change only to the XML tag structure, the textual content is unchanged. However, we can represent the change by deleting the word 'quick' and adding the element

```
<s>quick</s>
```

This is a perfectly valid representation of the change, but it implies that there has been deletion and addition and thus that the text has changed. This is shown below. The dx attribute indicates the documents in which the element and its content were present. The deltaxml:textGroup and deltaxml:text elements are wrappers introduced to delineate the word that has been deleted. We need the wrapper as a container for the dx attribute that applies to the text. The reason for the double wrapper here is that there may be more than one variant of the text, so more than one deltaxml:text element, and it is then useful to have these grouped in the outer deltaxml:textGroup for easier processing.

```
<p dx="A,B">The
   <deltaxml:textGroup dx="A">
      <deltaxml:text dx="A">quick</deltaxml:text>
   </deltaxml:textGroup>
   <s dx="B">quick</s>
   brown fox.
</p>
```

It would be preferable if we could represent this change without implying change to the content. This is discussed in the next section.

3. Representing Structural Change without Content Duplication

In order to avoid duplication of content, we need to distinguish between the element tag and its content so that we can make assertions about the tag and content separately and independently.

As a starting point, we can add an attribute to an element to indicate whether or not this element was present in a particular variant of the document. If the element was present, then the implication is that both the tag and the contents were present. In the above situation, we want to indicate that the content, i.e. the word 'quick', was present in two versions, but the tag, i.e. the <s>, was only present in one version. We can take a simple approach to this and add an additional attribute with this information.

```
<p dx="A,B" dxTag="A,B">The
```

---

[1] The delta format being used here is a simplified form of the DeltaXML DeltaV2.1 format [10 [162]]. The dx attribute would normally be a deltaxml:deltaV2 and the content would indicate whether or not the documents were the same or different for this element. This distinction is not important for this paper and so has been omitted to make the examples simpler.

```
  <s dx="A,B" dxTag="B">quick</s>
  brown fox.</p>
```

Here, the dx attributes tells us the documents in which the element (and its content) were present, as described above. But now the dxTag attribute tells us a bit more: whether or not the tag itself was present. So where the document identifiers are the same in both the dx attribute and the dxTag attribute, the element and its content were present. Where we see dx='A,B' and dxTag='B' we can deduce that the tag was present only in B. This means that A contained 'quick' and B contained '<s>quick</s>'.

We can optimize this a little by omiting the dxTag attribute if its value is the same as the dx value. Therefore we get:

```
<p dx="A,B">The
  <s dx="A,B" dxTag="B">quick</s>
  brown fox.</p>
```

This is a simple representation of a simple change. We can make an adjustment to this to represent, for example, a change from <i> in document A to <s> in document B as follows:

```
<p dx="A,B">The
  <i dx="A,B" dxTag="A"><s dx="A,B" dxTag="B">quick</s></i>
  brown fox.</p>
```

We can now introduce some overlap and see how the principles above are extended. When overlap occurs, in order to avoid duplicating content, we need to split some of the elements into fragments - this is the approach that Jeni Tennison calls 'fragmentation'. When we fragment an element, then clearly one original element becomes two or more fragments. The dxTag attribute refers to the whole tag, so we need to extend this to represent the start and the end. To achieve this we have dxTagStart and dxTagEnd so that we clearly distinguish between the start fragment and the end fragment. In more complex situations where an element is split into more than two fragments, we also introduce dxTagMiddle for any fragement betwen the start and end fragments.

This is an example of simple overlap:

```
<p>The quick brown fox. It jumped over the lazy dog.</p>
```

```
<p>The quick brown fox.</p><p> It jumped over the lazy dog.</p>
```

This is represented as:

```
<p dxTagStart="A" dxTag="B" dx="A,B">The quick brown fox.</p>
<p dxTagEnd="A" dxTag="B" dx="A,B"> It jumped over the lazy dog.</p>
```

This shows two <p> elements, and for the B document each of these represents a complete element, denoted by dxTag="B". For the A document, the two <p> elements are fragments and so the first is identified by dxTagStart="A" and the second one by dxTagEnd="A". This is an unambiguous representation that requires no duplication of textual content. The astute observer may comment that the leading space in the second paragraph of the B document would probably have been deleted. Proper handling of whitespace is a consumer of considerable time and effort in XML document processing. This type of change could be represented but it complicates the story so is ignored for this example.

We can now consider an example of double overlap, where text is moved from one paragraph to another:

```
<p>The quick brown fox. It jumped over the lazy dog.</p><p> Yes!</p>
```

```
<p>The quick brown fox.</p><p> It jumped over the lazy dog. Yes!</p>
```

This is represented as:

```
<p dxTagStart="A" dxTag="B" dx="A,B">The quick brown fox.</p>
<p dxTagEnd="A" dxTagStart="B" dx="A,B"> It jumped over the lazy dog.</p>
<p dxTag="A" dxTagEnd="B" dx="A,B"> Yes!</p>
```

This shows three <p> elements, all of which are fragments in at least one document. In the B document the first of these represents a complete element, denoted by dxTag="B". The last two <p> elements are fragments and so the first is identified by dxTagStart="B" and the second one by dxTagEnd="B". This mechanism will scale to any level of complexity,

for example three or more overlapping hierarchies. As overlap increases, so does the fragmentation and therefore the complexity of the result.

Although there is not time to explore this more fully in this paper, it would certainly be interesting to determine how easy it is to perform queries on this structure such as, "find all the paragraphs containing both the word 'fox' and the word 'dog'" and have this return just the A document because in the B document these words are in different paragraphs.

We can now look at a larger example including a change. We will for the example ignore white space changes. The A document is:

```
<book>
    <p>
        <seg>Scorn not the sonnet;</seg>
        <seg>critic, you have frowned, Mindless of its just honours;</seg>
        <seg>with this key SHAKESPEARE unlocked his heart;</seg>
        <seg>the melody Of this small lute gave ease to Petrarch's wound.</seg>
    </p>
</book>
```

And the second, B, document is as follows:

```
<book>
    <l>Scorn not the sonnet; critic, you have frowned,</l>
    <l>Mindless of its just honours; with this key</l>
    <l>Shakespeare unlocked his heart; the melody</l>
    <l>Of this small lute gave ease to Petrarch's wound.</l>
</book>
```

There are different representations that we can generate for this depending on how we decide to nest the fragments. For example, if we generally nest the <seg> elements inside the <l> elements, we get this result:

```
<book dx="A,B">
    <p dx="A,B" dxTag="A">
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTag="A">Scorn not the sonnet; </seg>
            <seg dx="A,B" dxTagStart="A">critic, you have frowned,</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">Mindless of its just honours; </seg>
            <seg dx="A,B" dxTagStart="A">with this key</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">
                <deltaxml:textGroup dx="A,B">
                    <deltaxml:text dx="A">SHAKESPEARE</deltaxml:text>
                    <deltaxml:text dx="B">Shakespeare</deltaxml:text>
                </deltaxml:textGroup> unlocked his heart;</seg>
            <seg dx="A,B" dxTagStart="A">the melody</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">Of this small lute gave ease to Petrarch's
                wound.</seg>
        </l>
    </p>
</book>
```

It is instructive to visualize this structure as shown below. Here we are looking at it primarily as document A, so the tags and text that belong only to B have been greyed out. This is to visualize more clearly the A structure. Some of the <seg> elements are still split so these would need to be

merged in order to get back to the original A document, although the basic original structure of A is

```
<book dx="A,B">
    <p dx="A,B" dxTag="A">
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTag="A">Scorn not the sonnet; </seg>
            <seg dx="A,B" dxTagStart="A">critic, you have frowned,</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">Mindless of its just honours; </seg>
            <seg dx="A,B" dxTagStart="A">with this key</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">
                <deltaxml:textGroup dx="A,B">
                    <deltaxml:text dx="A">SHAKESPEARE</deltaxml:text>
                    <deltaxml:text dx="B">Shakespeare</deltaxml:text>
                </deltaxml:textGroup>
                unlocked his heart;</seg>
            <seg dx="A,B" dxTagStart="A">the melody</seg>
        </l>
        <l dx="A,B" dxTag="B">
            <seg dx="A,B" dxTagEnd="A">Of this small lute gave ease to Petrarch's
                wound.</seg>
        </l>
    </p>
</book>
```

apparent.

This visualization illustrates the very simple nature of this approach. The attributes we are adding provide information about an element, specifically for each variant the attributes tell us which of the following is true:

· The tag and its content are present in this variant and the element is not fragmented

· The tag and its content are present in this variant and the element is fragmented, so this is the start, the end or a middle fragment

· The content is present in this variant but not the tag

· The tag and its content are not present in this variant

Therefore it is very simple to extract any one variant from the whole document or any part of it. It is also very simple to work out, for a given piece of content, the list of ancestors in any variant. An important characteristic of this representation is that as the overlap reduces to zero so the representation reduces to the original structure.

4. Dominant Hierarchy

Methods for representing overlapping hierarchy often need to know the dominant hierarchy in order to know which tree structure 'overrides' the others. In this proposed representation, there is no need for a concept of a dominant hierarchy. We are at liberty to create a hierarchy that reduces the fragmentation as far as possible. Therefore it is possible to adopt various different algorithms to generate different results. The format describes how to represent overlapping hierarchy, it does not dictate what the overlap should be. Therefore another valid representation of the example above would be as follows:

```
<book xmlns:dx="xx" dx="A,B">
    <p dx="A,B" dxTag="A">
        <seg dx="A,B" dxTag="A">
            <l dx="A,B" dxTagStart="B">Scorn not the sonnet;</l>
        </seg>
        <seg dx="A,B" dxTagStart="A">
            <l dx="A,B" dxTagEnd="B">critic, you have frowned, </l>
        </seg>
```

```
            <seg dx="A,B" dxTagEnd="A">
                <l dx="A,B" dxTagStart="B">Mindless of its just honours;</l>
            </seg>
            <seg dx="A,B" dxTag="A">
                <l dx="A,B" dxTagEnd="B">with this key </l>
                <l dx="A,B" dxTagStart="B">
                    <dx:textGroup dx="A,B">
                        <dx:text dx="A">SHAKESPEARE</dx:text>
                        <dx:text dx="B">Shakespeare</dx:text>
                    </dx:textGroup>
                    unlocked his heart;</l>
            </seg>
            <seg dx="A,B" dxTag="A">
                <l dx="A,B" dxTagEnd="B">the melody </l>
                <l dx="A,B" dxTag="B">Of this small lute gave ease to Petrarch's
                    wound.</l>
            </seg>
        </p>
</book>
```

We can also take this a step further, and look at the representation for what might be called full fragmentation, i.e. each piece of text that has a different set of ancestors is put into a single fragment. It would also be possible to treat the paragraph element in the same way, but ideally this can be kept as a single element around all of the text, providing a clearer and simpler representation.

```
<book dx="A,B">
    <p dx="A,B" dxTag="A">
        <seg dx="A,B" dxTag="A">
            <l dx="A,B" dxTagStart="B">Scorn not the sonnet;</l>
        </seg>
        <seg dx="A,B" dxTagStart="A">
            <l dx="A,B" dxTagEnd="B">critic, you have frowned, </l>
        </seg>
        <seg dx="A,B" dxTagEnd="A">
            <l dx="A,B" dxTagStart="B">Mindless of its just honours;</l>
        </seg>
        <seg dx="A,B" dxTagStart="A">
            <l dx="A,B" dxTagEnd="B">with this key </l>
        </seg>
        <seg dx="A,B" dxTagEnd="A">
            <l dx="A,B" dxTagStart="B">
                <dx:textGroup dx="A,B">
                    <dx:text dx="A">SHAKESPEARE</dx:text>
                    <dx:text dx="B">Shakespeare</dx:text>
                </dx:textGroup>
                unlocked his heart;</l>
        </seg>
        <seg dx="A,B" dxTagStart="A">
            <l dx="A,B" dxTagEnd="B">the melody </l>
        </seg>
        <seg  dx="A,B" dxTagEnd="A">
            <l dx="A,B" dxTag="B">Of this small lute gave ease to Petrarch's wound.</l>
        </seg>
    </p>
</book>
```

The actual hierarchy of the overlapping elements can be determined based on any criteria. One criterion might be to minimise the fragmentation. The results of an automated generation of the above by comparing the two documents and aligning them according to their text content is shown below. In this example the attribute names are shown in full, e.g. dx attribute is shown as deltaxml:deltaV2 and its content indicates whether the two documents are equal, i.e. "A=B" or not equal, i.e. "A!=B". The hierarchy is reconstructed to reduce fragmentation.

```
<book xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
```

```
            deltaxml:deltaV2="A!=B"
            deltaxml:version="2.1" deltaxml:content-type="full-context">
            <p deltaxml:deltaV2="A!=B" deltaxml:deltaTag="A">
                <seg deltaxml:deltaV2="A!=B" deltaxml:deltaTag="A">
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagStart="B"
                        >Scorn not the sonnet;</l>
                </seg>
                <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagMiddle="B"> </l>
                <seg deltaxml:deltaV2="A!=B" deltaxml:deltaTag="A">
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagEnd="B">critic,
                    you have frowned,</l>
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagStart="B">Mindless
                    of its just honours;</l>
                </seg>
                <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagMiddle="B"> </l>
                <seg deltaxml:deltaV2="A!=B" deltaxml:deltaTag="A">
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagEnd="B">with this key</l>
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagStart="B">
                        <deltaxml:textGroup deltaxml:deltaV2="A!=B">
                            <deltaxml:text deltaxml:deltaV2="A"
                                >SHAKESPEARE</deltaxml:text>
                            <deltaxml:text deltaxml:deltaV2="B"
                                >Shakespeare</deltaxml:text>
                        </deltaxml:textGroup>
                        unlocked his heart;</l>
                </seg>
                <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagMiddle="B"> </l>
                <seg deltaxml:deltaV2="A!=B" deltaxml:deltaTag="A">
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTagEnd="B">the melody</l>
                    <l deltaxml:deltaV2="A!=B" deltaxml:deltaTag="B">Of this
                        small lute gave ease to Petrarch's wound.</l>
                </seg>
            </p>
        </book>
```

In addition there are several elements that contain only white space, e.g. the second <l> element. This is because the A document contained a space between the two <seg> elements:

```
<seg>Scorn not the sonnet;</seg> <seg>critic, you have frowned,
Mindless of its just honours;</seg>
```

The B document had this space within the <l> element:

```
<l>Scorn not the sonnet; critic, you have frowned,</l>
```

As mentioned earlier, correct handling of white space is often very complicated because a careful distinction needs to be made between white space that can be ignored and white space that is part of the content. Element boundaries are not always word separators, for example elements that represent formatting are not considered word separators whereas a new line would be considered a word separator. This is often not clearly specified or represented in the XML schema.

The overlapping hierarchy representation described here is therefore suited to a number of different situations.

5. Attributes

Attributes are an important part of the XML structure, and have not yet been mentioned. Where an element appears in a particular document variant, and is not fragmented, it is simple to add the attributes onto that element as part of the start tag. When an element has been fragmented, then the attributes for that element will appear in the start tag, i.e. the element with the dxTagStart attribute. This means that any attributes that appear on a middle tag or end tag would not be relevant to a particular document variant.

This is an example of simple overlap including some attribute data:

```
<p>The quick brown fox. It jumped over the lazy dog.</p>
```

```
<p>The quick brown fox.</p><p class="B"> It jumped over the lazy dog.</p>
```

This is represented as:

```
<p dxTagStart="A" dxTag="B" dx="A,B">The quick brown fox.</p>
<p dxTagEnd="A" dxTag="B" dx="A,B" class="B"> It jumped over the lazy dog.</p>
```

This shows the class attribute but an attribute applies only to those variants where the tag is a dxTag or dxTagStart. Therefore class="B" applies only to the B document because for A this <p> is an end tag.

Changes to attributes can also be represented. This is done by converting the attribute into markup as part of a new first child of the element. Although theoretically possible to represent changes to attributes within attributes, this leads to some dedicated syntactic conventions within the attribute string, which is not easy to process. Therefore separating change attributes out into XML markup makes processing, particularly using XSLT, much easier.

This is an example of simple overlap, including some changed attribute data:

```
<p class="B" align="left">The quick brown fox. It jumped over the lazy dog.</p>
```

```
<p class="B" align="right">The quick brown fox.</p><p> It jumped over the lazy dog.</p>
```

This is represented as:

```
<p dxTagStart="A" dxTag="B" dx="A,B" class="B">
  <deltaxml:attributes>
    <dxa:align dx="A,B">
      <deltaxml:attributeValue dx="A">left</deltaxml:attributeValue>
      <deltaxml:attributeValue dx="B">right</deltaxml:attributeValue>
    </dxa:align>
  </deltaxml:attributes>
The quick brown fox.</p>
<p dxTagEnd="A" dxTag="B" dx="A,B"> It jumped over the lazy dog.</p>
```

This shows that the unchanged attribute, class="B", remains as an attribute, but the changed align attribute is represented as markup to show the two values. This is a simplified representation and full details can be found in the documentation of the DeltaXML DeltaV2.1 format [11 [162]].

The delta representation also allows an alternative representation because the <p> tag in the A document can be wrapped around the two <p> tags in the B document, as shown below:

```
<p dxTag="A" dx="A,B" class="B" align="left">
 <p dxTag="B" dx="A,B" class="B" align="right">The quick brown fox.</p>
 <p dxTag="B" dx="A,B"> It jumped over the lazy dog.</p>
</p>
```

This is, in this case, a shorter representation though it has in effect used duplication of the (unchanged) attributes and tags to show the change. However, this may be a preferred representation for some formatting elements, for example if the class attribute in a <span> is changed then it may be more useful to represent this as a different <span>. Both representations conform to the delta format.

6. Processing Observations

This representation requires that some elements are split, although there is no requirement for a dominant hierarch as such. A given overlap situation can be represented in a number of different ways, all of which are valid in that they correctly represent the overlap. As mentioned earlier, a representation is deemed to be correct if it is possible to generate each of the original hierarchies without loss from the overlap representation, and this implies there may be more than one correct representation. There may be different ways to define what might be an 'optimal' representation, but it seems that minimising the splitting of elements is a key aspect of this.

Some form of milestone representation may be a starting point because this is one of the most intuitive ways to represent overlap. How then do we process a milestone representation to get to an optimal solution using the delta format described here? The problem is to work out where there is overlap and which element or elements need to be split to remove this overlap. This is fairly easy for simple overlaps of two elements but it is not simple in the situation where there are multiple, arbitary overlaps. Minimising splits can result in having different XML hierarchies, e.g. a particular element

type in document A may be both surrounding and within the same element type from document B. Both the A and B document can be generated from this but it can look odd in the delta file and a processor cannot rely on the same nesting in all situations. It is also true that although elements such as <b> or <i> could be nested either one inside the other (the nesting is commutative), this may not be the case with all element combinations.

In practice, different representations are useful for different purposes. In some cases it is preferable to avoid overlap by duplication of content but then the issue is to find the minimum duplication to achieve this. It may be easy in some situations to process milestones but again it is preferable that milestones are only used when they need to be and by finding the minimul split representation it is simple then to achieve minimum milestones from this.

7. Conclusions

This paper has described a representation for overlapping hierarchy which is also capable of representing changes to text and attributes. This makes it suitable for some important use cases for overlapping hierarchy, particularly the representation of change between two or more variants of a document.

A significant advantage over some previous representations is that it is pure XML, and therefore can be processed using standard XML tools. The dominance of one hierarchy over another does not need to be fixed and this means that the actual hierarchy of the overlapping structures can be determined for other reasons and indeed varied throughout the document. This flexibility allows fragmentation of elements to be kept to a minimum.

The underlying data model is based on document variants and therefore is better suited to situations where the number of variants is small. Although it does scale to any number of variants, its complexity increases as the number of variants increases, e.g. each new variant has an identifier in the dx attribute so this will become longer and more difficult to interpret.

Overlapping hierarchy is a powerful tool to use in certain markup situations, though its use can lead to complex situations and any solution is also likely to look complicated. This paper is intended to contribute to the discussion as the XML community continues to strive for a simple, generic and universal solution to this problem.

An earlier version of this paper was presented at Balisage 2016 [12 [163]].

References

[1] Overlap, Containment and Dominance. URN: "http://www.jenitennison.com/2008/12/06/overlap-containment-and-dominance.html"

[2] Modeling overlapping structures, Yves Marcoux, Michael Sperberg-McQueen, Claus Huitfeldt, URN: "http://www.balisage.net/Proceedings/vol10/html/Marcoux01/BalisageVol10-Marcoux01.html"

[3] Markup Overlap: A Review and a Horse, Steven DeRose, URN: "http://conferences.idealliance.org/extreme/html/2004/DeRose01/EML2004DeRose01.html"

[4] Multiple hierarchies: new aspects of an old solution, Andreas Witt, URN: "http://conferences.idealliance.org/extreme/html/2004/Witt01/EML2004Witt01.html" />

[5] Representation of overlapping structures, Michael Sperberg-McQueen, URN: "http://conferences.idealliance.org/extreme/html/2007/SperbergMcQueen01/EML2007SperbergMcQueen01.html"

[6] TEI: Text Encoding Initiative, URN: "http://www.tei-c.org/index.xml"

[7] OASIS Darwin Information Typing Architecture (DITA) TC, URN: "https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=dita"

[8] OASIS DocBook TC, URN: "https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=docbook" />

[9] Schmidt, Desmond. "Merging Multi-Version Texts: a Generic Solution to the Overlap Problem." Balisage Series on Markup Technologies, vol. 3 (2009), URN: "http://www.balisage.net/Proceedings/vol3/html/Schmidt01/BalisageVol3-Schmidt01.html"

[10] Overlapping Hierarchies in DeltaV2 Format , URN: "http://www.deltaxml.com/support/documents/deltav21"

[11] Two and Three Document DeltaV2 Format (patent pending), URN: "http://www.deltaxml.com/support/documents/deltav2"

[12] La Fontaine, Robin, "Representing Overlapping Hierarchy as Change in XML", Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016, URN: "https://doi.org/10.4242/BalisageVol17.LaFontaine01"

The XForms 2.0 Test Suite

Steven Pemberton, CWI, Amsterdam

Abstract

XForms 1.0 and 1.1 both had test suites that consisted largely of static XForms documents. To run the tests you had to manually activate them one by one, and then visually confirm that the output matched the description of what should have been produced. If you wanted to add more cases to a test, it involved adding to the set of documents, or editing the individual documents.

The test suite for XForms 2.0 now being constructed takes a different approach, the idea being that the tests should check themselves that they have passed; most tests have a similar structure so that only the data used needs to be altered to check new cases.

Of course, for a language designed for user-interaction, some tests have to be based on physical interaction. But once you have confirmed that clicking on a button does indeed generate the activation event, all subsequent tests can generate the activation event without user intervention.

The introspection needed for tests to check the workings of the processor doing the testing can raise some challenging problems, such as how to test that the initial start-up event has been sent when the facilities for recording that fact have not yet been initialised.

This paper considers the techniques used to create a self-testing XForms test suite, some of the problems encountered, and gives examples of how some of them were solved.

1. Introduction

XForms [1] is a W3C standard XML-based markup language which, as the name would lead you to expect, was originally designed for a new generation of forms on the Web. Indeed, version 1 was exactly that, and to a large extent mirrored what could be achieved with HTML forms, while adding extra facilities. However, after a short period of experience, it was realised that with a small amount of generalisation, XForms would be suitable for other sorts of interactive applications as well.

And so was born XForms 1.1, a declarative, Turing-complete programming language, applicable for interactive applications (and forms) both on and off the web.

Since then XForms has been used by a broad, international user population, from small companies to multinationals, with more than six implementations available from around the world.

One of the surprises has been that experience has repeatedly shown that using XForms for applications reduces application development time by an order of magnitude, with concomitant reductions in cost. This is largely thanks to the declarative style of programming that XForms uses: much of the administrative side of regular procedural programming is taken care of automatically by the XForms system.

Now, XForms 2.0 is in development [2], a further generalisation of the previous version.

2. Introduction to XForms

The essence of XForms is *state*, which means that it meshes extremely well with the REST (Representational State Transfer) architectural style [3].

An XForms application has two parts: a *model*, and a *user-interface*.

The model contains all the data being used, stored in *instances*, along with descriptions of properties of, and relationships between, the data. For instance, data can be defined inline:

```
<instance>
   <tests xmlns="">
      <test pass="" res="" req="valid">2018-01-20</test>
      <test pass="" res="" req="invalid">2018/01/20</test>
   </tests>
</instance>
```

or can be imported from an external source:

```
<instance src="tests.xml"/>
```

Descriptions of data properties are done using *bind* statements that bind properties to data values:

```
<bind ref="today" type="date"/>
<bind ref="color" readonly="../variant='basic'"/>
<bind ref="state" required="../country='USA'"/>
<bind ref="state" relevant="../country='USA'"/>
<bind ref="total" calculate="../number * ../unitprice"/>
<bind ref="height" constraint=". > 0"/>
```

Controls in the user-interface then *bind* to the data to allow interaction:

```
<input ref="age" label="Age:"/>
```

After initialisation the system is in *stasis*: the data matches the descriptions, and the relationships between data are up-to-date. After that, *events* occur, either system-generated or user-initiated, causing data values to change, to which the XForms system responds in order to return the system to stasis. While in general the system responds to events in standard ways, applications can also catch *events* and specify *actions* that define how to respond to particular events in particular ways:

```
<action ev:event="xforms-value-changed">
   <setvalue ref="unsaved">true</setvalue>
```

```
</action>
```

As a --simple-- example, a map application might keep the x and y coordinates of a location and a zoom level for looking at the map:

```
<instance>
    <map xmlns="">
        <zoom>10</zoom>
        <x>511</x>
        <y>340</y>
        <url/>
    </map>
</instance>
```

The URL of the relevant map tile for that location can then --automatically-- be kept up-to-date whenever and however any of the values change, by using a bind:

```
<bind ref="url"
      calculate="concat('http://tile.openstreetmap.org/',
                        ../zoom, '/', ../x, '/', ../y, '.png')"/>
```

See [4] for a further introduction to XForms, and [5] for a fully worked-out mapping application in XForms.

3. Test Suites

As part of the process of defining a new standard, it is recognised good practice to define a test suite to go along with the specification. The principle reason for having a test suite of course is to allow implementers to check that their implementations correctly interpret the definitions in the specification. However, users of implementations can gain confidence in the implementation they use by seeing the results from the test suite. And finally, it is useful for the specification writers themselves: by forcing them to think of test examples, it can expose corners of the specification that have not yet been sufficiently well defined.

4. The XForms 1.* test suite

The original XForms 1.0 and 1.1 test suites [6] were defined as a large collection of files, each file testing one feature of the language. Each test consisted of some output, plus a description of what you should see if the test had passed. Problems experienced with this approach included:

- It was tedious to run each test one by one,

- It was concentrated work deciding if a test had passed,

- Adding tests involved authoring a complete file for each test.

5. The XForms 2.0 test suite

The problems experienced with the earlier versions of the test suite led us to approach the test suite for the new version of XForms in a totally different way.

Firstly, the test suite is now One Big XForm, that loads the details of the tests into an XForms instance, and uses that to drive the test suite interface, running the tests as sub-forms . The tests are divided over the chapters of the specification, and each chapter contains a test case for each feature, each test case containing any number of tests for that feature.

The interface allows you to step through the chapters, or to select a particular chapter, and for each chapter to step through the tests, or select a particular test.

Secondly, as far as possible, the tests are all self testing: apart from a description of what is being tested, and the output of the results, tests include a big green PASS or a big red FAIL indication at the top of the output, signifying whether the output values match with what was expected, with each individual failing test within that one test case also being flagged.

However, not all tests can be self-testing. Of course, for a language designed for user-interaction, some tests have to be based on physical interaction: for instance, does clicking on a button generate the correct event? But once you have confirmed it does indeed, subsequent tests can generate the activation event without user intervention.

Similarly, some tests can only be confirmed by inspection: does the `now()` function indeed generate today's date and time? Once you have confirmed that, other tests that depend on the date and time can use that function without further inspection.

Some tests are particularly introspective. Are expressions evaluated in the correct evaluation context? Such tests are particularly hard to formulate, since the very act of introspection alters the context that they are being evaluated in.

Finally, testing initialisation can be tricky, because before the system has initialised, there is little you can do.

6. The Generic Structure of the Tests

Despite the caveats above regarding tests that cannot be tested automatically, the vast majority can, and almost all use a standard template.

To demonstrate the workings of this template, let us consider an example test case for a function, in this case for the function `boolean-from-string()`.

We want to test that a function calls such as `boolean-from-string('true')` return the correct result.

To do this, the parameter value is enclosed in an element:

```
<test>true</test>
```

and we add attributes where the required result, the actual result, and whether the test case passes or not will be stored:

```
<test pass="" res="" req="true">true</test>
```

As many such test cases as necessary are then gathered together in an instance:

```
<instance>
   <tests pass="" name="boolean-from-string() function" xmlns="">
      <test pass="" res="" req="true">true</test>
      <test pass="" res="" req="true">TRUE</test>
      <test pass="" res="" req="true">tRue</test>
      <test pass="" res="" req="true">1</test>
      <test pass="" res="" req="false">false</test>
      <test pass="" res="" req="false">FALSE</test>
      <test pass="" res="" req="false">faLse</test>
      <test pass="" res="" req="false">0</test>
      <test pass="" res="" req="false">qwertyuiop</test>
      <test pass="" res="" req="false"></test>
      ...
   </tests>
</instance>
```

A bind is then used to calculate the individual results:

```
<bind ref="test/@res" calculate="boolean-from-string(..)"/>
```

whose effect is to calculate the `res` attribute for all `test` elements.

Another bind, independent of which function is being tested, calculates if the computer result matches the expected value:

```
<bind ref="test/@pass" calculate="if(../@res = ../@req, 'yes', 'no')"/>
```

and finally a bind for the attribute on the outmost element records if all tests have passed:

```
<bind ref="@pass" calculate="if(//test[@pass!='yes'], 'FAIL', 'PASS')"/>
```

which says that if there is a `test` element whose `pass` attribute does not have the value `yes`, then the test set fails, and otherwise it passes. We may in future also add a percentage pass value, that counts the number of passed tests:

```
<bind ref="@percent" calculate="100*count(//test[@pass='yes']) div count(//test)"/>
```

With this structure, every test form has an identical set of controls, that output the name of the test, an optional description (which, following XForms rules, is only displayed if present in the instance), whether all tests have passed, for quick inspection, and the list of each test with an indication of what was expected when it has failed:

```
<group>
   <label class="title" ref="@name"/>
   <output class="block" ref="description"/>
   <output class="{@pass}" ref="@pass"/>
   <repeat ref="test">
      <output value="."/> → <output ref="@res"/>
      <output class="wrong" ref="@req[.!=../@res]"/>
   </repeat>
</group>
```

Note that with the statement

```
<output class="wrong" ref="@req[.!=../@res]"/>
```

this only selects the `req` attribute if its value does not match that of the `res` attribute on the same element. If they match, the `req` is not selected, and nothing is output.

This looks like this when run:

boolean-from-string() function

PASS

true → true
TRUE → true
tRue → true
1 → true
false → false
FALSE → false
faLse → false
0 → false
test → false
 → false

Here is an example of a fail (and in this case with a description as well):

seconds-from-epoch() function

The exact same semantic as seconds-from-dateTime but under a name which doesn't clash with the XPath 2.0 function of the same name.

FAIL

```
1970-01-01T00:00:00Z → 0
1970-01-01T00:00:00 → 0
1969-12-31T23:59:59 → -1
1970-01-01T00:00:00-08:00 → -2880000   expected: 28800
1970-01-01T00:01:00+00:01 → 120   expected: 0
1970-01-01T00:00:00-00:01 → -60
1970-01-01T00:00:00-00:02 → -120
1970-01-01T00:01:00-00:02 → -60
1970-01-01T00:02:00-00:02 → 0
1970-01-01T00:01:00-00:01 → 0
1970-01-01T00:00:00-01:00 → -360000   expected: -3600
1970-01-01T00:02:00+00:02 → 240   expected: 0
1970-01-01T01:00:00+01:00 → 363600   expected: 0
1970-01-01T02:00:00+02:00 → 727200   expected: 0
1970-01-01T00:00:00.001Z → 0.001
1970-01-01T00:00:01Z → 1
```

If we want to test a function with more than one parameter, we structure the `test` elements slightly differently, for instance for the `compare()` function which has two parameters, by enclosing each parameter in an element of its own:

```
<test pass="" res="" req="-1">compare(<a>apple</a>, <b>orange</b>)</test>
```

and then modify the bind that calculates the results:

```
<bind ref="test/@res" calculate="compare(../a, ../b)"/>
```

Note that in the general template structure, these are the only places where there are differences between test cases: in the data, and in the bind calculating the result. The rest remains identical.

7. Datatypes

To test datatypes, we want to do something similar. We can collect in the `test` elements a, possibly large, group of values of the datatype to be tested, and then see if the system thinks they are valid values for that type. For instance for the `date` datatype:

```
<test pass="" res="" req="invalid"/>
<test pass="" res="" req="valid">2018-01-20</test>
<test pass="" res="" req="invalid">2018/01/20</test>
<test pass="" res="" req="valid">-2018-01-20</test>
<test pass="" res="" req="invalid">+2018-01-20</test>
<test pass="" res="" req="invalid">02018-01-20</test>
<test pass="" res="" req="valid">12018-01-20</test>
```

and so on, and then calculate with the bind for the results:

```
<bind ref="test/@res" calculate="if(valid(.), 'valid', 'invalid')"/>
```

This would work fine, but for our purposes it has a difficulty. The `valid()` function is an XForms 2.0 addition, and we want as litttle as possible of the test suite infrastructure to depend on XForms 2.0 features -- if anything is likely to fail it is the new features of the language before the old features; all the more so since many of the tests will still work with older versions of XForms and so can still be used on older implementations.

In XForms 1.1 (and later), if a control is bound to a value, and its value changes, an event is dispatched to the control announcing its validity. The standard XForms response is to change the display of the value to make it clear that it is now newly valid or invalid, but we can catch the event and record that it has happened for that value. The only difficulty that we have to deal with is that the event is only generated when the value changes.

So what we do is initially set the test value to some random value, it doesn't matter what it is, nor whether it is a valid or invalid value for the datatype, and when the system has initialised, only then change all the values to the data we are actually interested in. Then when the value changes, and the event is generated, we catch it and save the result. Something along these lines:

Store the value we are interested in in an attribute:

```
<test pass="" res="" req="valid" val="2018-01-20"/>
```

Add an attribute to the root element to record whether the system has been initialised yet:

```
<tests pass="" started="" name="date type" xmlns="">
```

And then use a bind to calculate the value of the elements:

```
<bind ref="test" type="date" calculate="if(../@started='', 'xxx', @val)"/>
```

This ensures that initially the `test` elements have the value 'xxx', until the `started` attribute is changed, which we do on initialisation, by catching the `xforms-ready` event:

```
<action ev:event="xforms-ready">
   <setvalue ref="@started">yes</setvalue>
</action>
```

Then in the output section, we can catch the validity events, and record them:

```
<repeat ref="test">
   <output ref=".">
      <action ev:event="xforms-valid">
         <setvalue ref="@res">valid</setvalue>
      </action>
      <action ev:event="xforms-invalid">
         <setvalue ref="@res">invalid</setvalue>
      </action>
   </output>
   ...
```

Which might look like this:

```
date type

FAIL

:  → valid    expected: invalid
20/01/2018: 2018-01-20 → valid
2018/01/20: 2018/01/20 → invalid
-2018-01-20: -2018-01-20 → invalid    expected: valid
+2018-01-20: +2018-01-20 → invalid
02018-01-20: 02018-01-20 → invalid
12018-01-20: 12018-01-20 → invalid    expected: valid
```

8. Events

In the previous example, we saw some events that happen during processing: the `xforms-ready` event that is dispatched when the system has finished initalising, and the `xforms-valid` and `-invalid` events that are dispatched when a value bound to a control changes.

In fact, when such a value changes several states are announced via events: whether the control is enabled or not, whether the value is optional or required, whether the value is readonly or not, as well as the two we have already seen. The test to check that these events are sent correctly starts by assembling test values that are all zero:

```
<test pass="" res="" req="disabled">0</test>
<test pass="" res="" req="enabled">0</test>
<test pass="" res="" req="optional">0</test>
<test pass="" res="" req="required">0</test>
<test pass="" res="" req="readwrite">0</test>
<test pass="" res="" req="readonly">0</test>
<test pass="" res="" req="valid">0</test>
<test pass="" res="" req="invalid">0</test>
```

and binding to the values properties, so that each positive property (such as valid) is the case when the value is 1, and the opposite property (such as invalid) is the case when the value is 0:

```
<bind ref="test/@req[.='enabled']"    relevant="..=1"/>
<bind ref="test/@req[.='disabled']"   relevant="..=0"/>
<bind ref="test/@req[.='valid']"      constraint="..=1"/>
<bind ref="test/@req[.='invalid']"    constraint="..=0"/>
<bind ref="test/@req[.='required']"   required="..=1"/>
<bind ref="test/@req[.='optional']"   required="..=0"/>
<bind ref="test/@req[.='readonly']"   readonly="..=1"/>
<bind ref="test/@req[.='readwrite']" readonly="..=0"/>
```

When initialisation is finished, all the test values are flipped to 1:

```
<action ev:event="xforms-ready">
   <setvalue iterate="test" ref=".">1</setvalue>
</action>
```

which causes all the properties to flip. The resultant events are then caught in the output section:

```
<repeat ref="test">
   <output ref="@req"><label>Event</label>
      <setvalue ref="../@res" ev:event="xforms-disabled" value="concat(., 'disabled')"/>
      <setvalue ref="../@res" ev:event="xforms-enabled" value="concat(., 'enabled')"/>
      <setvalue ref="../@res" ev:event="xforms-optional" value="concat(., 'optional')"/>
      <setvalue ref="../@res" ev:event="xforms-required" value="concat(., 'required')"/>
      <setvalue ref="../@res" ev:event="xforms-readwrite" value="concat(., 'readwrite')"
      <setvalue ref="../@res" ev:event="xforms-readonly" value="concat(., 'readonly')"/>
      <setvalue ref="../@res" ev:event="xforms-valid" value="concat(., 'valid')"/>
      <setvalue ref="../@res" ev:event="xforms-invalid" value="concat(., 'invalid')"/>
   </output>
   ...
```

Note that by concatenating the result, we catch the case where the event is (incorrectly) sent more than once.

Here is an example of the output:

MIP Notification Events

Check the notification events xforms-disabled, -enabled, -optional, -required, -readwrite, -readonly, -valid, -invalid. These events are only sent when the state changes, so initially all fields are in the opposite state, and then when the form is ready, the states are flipped.

PASS

→ disabled
Event
enabled → enabled
Event
optional → optional
Event*
required → required
Event
readwrite → readwrite
Event
readonly → readonly
Event
valid   → valid
Event ✖
invalid   → invalid

9. Actions

Actions often cause a change to the data. A good example of such an action is `insert`, that inserts new elements or attributes into a data structure.

After an insert, the system has to restore stasis, and goes through a number of steps to do that: *rebuild*: possibly updating internal data structures, *recalculate*: recalculating dependent values, *revalidate*: checking changed values for validity, *refresh*: updating the user interface.

At each step of restoring stasis an event is dispatched. Although seldom needed, these events allow applications to do extra steps if necessary.

Doing extra processing during these stages requires care, because, by definition, the system is not yet up to date. In particular, changing values during these stages necessitates you manually doing an extra *recalculate* afterwards, since the system may not be aware of the changes you have made. It also means for instance that we can't keep an index of how many events received, and use that to index into a list of events received.

So what we do, is start off with the test instance containing elements for the expected events, except the very last (refresh):

```
<test pass="" res="" req="insert"/>
<test pass="" res="" req="rebuild"/>
<test pass="" res="" req="recalculate"/>
<test pass="" res="" req="revalidate"/>
```

then on `xforms-ready`, we use `insert` to add the missing element. With no further parameters, an `insert` on a list just duplicates the last element, so we need to update the `req` attribute:

```
<action ev:event="xforms-ready">
    <insert ref="test"/>
    <setvalue ref="test[last()]/@req">refresh</setvalue>
</action>
```

Then we catch all the events we are expecting, and store them at the locations they should be in if the events come in in the right order:

```
<action ev:event="xforms-insert">
    <setvalue ref="test[1]/@res">insert</setvalue>
</action>
<action ev:event="xforms-rebuild">
    <setvalue
    ref="test[@res='insert']/following-sibling::test[1]/@res[.='']">rebuild</setvalue>
</action>
<action ev:event="xforms-recalculate">
    <setvalue
    ref="test[@res='rebuild']/following-sibling::test[1]/@res[.='']">recalculate</setvalu
</action>
<action ev:event="xforms-revalidate">
    <setvalue
    ref="test[@res='recalculate']/following-sibling::test[1]/@res[.='']">revalidate</setv
</action>
<action ev:event="xforms-refresh">
    <setvalue
    ref="test[@res='revalidate']/following-sibling::test[1]/@res[.='']">refresh</setvalue
    <recalculate/>
</action>
```

A `setvalue` such as

```
<setvalue
  ref="test[@res='rebuild']/following-sibling::test[1]/@res[.='']">recalculate</setvalue
```

selects the `test` element after the one whose `res` attribute is `rebuild`, and sets the value of its `res` attribute only if it has not already been set. Therefore, if the rebuild event has not yet been received, we won't record the recalculate.

10. Initialisation

The big obstacle to testing initialisation is that during initialisation almost nothing is available: you are unable to use instance values, or calculations. The original 1.* test suite just displayed a dialogue box to announce that the initialisation events had been received:

```
<message ev:event="xforms-model-construct">xforms-model-construct received</message>
```

However, in the version 2 test suite, we have succeeded in finding a way to record that the event happened, in an instance value, so that the test can self-check.

When we receive the initialisation event, all we do is dispatch a new event with a delay long enough to allow initialisation to complete (1000 milliseconds is more than enough):

```
<action ev:event="xforms-model-construct">
   <dispatch name="yes" delay="1000" targetid="M"/>
</action>
```

When the new event is caught, initialisation will have finished, and we can then record that the initial event was seen:

```
<action ev:event="yes">
   <setvalue ref="test[1]">xforms-model-construct</setvalue>
</action>
```

11. Conclusion

We wanted to create a test suite that was easy to use, easy to create tests for, and easy to update, and we are very happy with the results so far. This is still work in progress; we estimate that about half the tests have been written to date. However, we already have good experience with the suite, and its easy modifyability helps us with constructing it.

Future work, apart from writing the remaining tests, will look at the possibilities of running the tests automatically, with minimum human intervention.

Bibliography

[1] *XForms 1.1.* John M. Boyer. W3C. 2009. https://www.w3.org/TR/xforms/ .

[2] *XForms 2.0.* Erik Bruchez et al.. W3C. 2018. https://www.w3.org/community/xformsusers/wiki/XForms_2.0 .

[3] *"Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.)..* Roy Thomas Fielding. University of California, Irvine.. 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm .

[4] *XForms: an Introduction, .* Steven Pemberton. CWI, Amsterdam. 2018. https://www.cwi.nl/~steven/xforms/ .

[5] *XForms: an Unusual Worked Example.* Steven Pemberton. CWI, Amsterdam. 2015. https://www.cwi.nl/~steven/Talks/2015/11-05-example/ .

[6] *XForms Test Suites.* Steven Pemberton. W3C. 2008. https://www.w3.org/MarkUp/Forms/Test/ .

[7] *XForms 2.0 Test Suite.* Steven Pemberton. CWI, Amsterdam. 2018. https://www.cwi.nl/~steven/forms/TestSuite/index.xhtml .