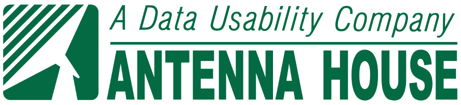
Markup UK

2023 Proceedings

XIOS3
Cloud//Backend

FUSIONDB


A Data Usability Company
ANTENNA HOUSE

le-tex
publishing services

<oXygen/>®
XML Editor

SAXONICA.COM
XSLT AND XQUERY PROCESSING

Organisation Committee

Geert Bormans
Ari Nordström
Andrew Sales
Rebecca Shoob

Programme Committee

Syd Bauman – Northeastern University Digital Scholarship Group
Achim Berndzen – <xml-project />
Tony Graham – Antenna House
Michael Kay – Saxonica
Jirka Kosek – University of Economics, Prague
Deborah A. Lapeyre – Mulberry Technologies
David Maus – State and University Library Hamburg
Adam Retter – Evolved Binary
B. Tommie Usdin – Mulberry Technologies
Norman Walsh – MarkLogic
Lauren Wood – XML.com

Thank You

CloudBackend
Evolved Binary
Antenna House
le-tex Publishing Services
oXygen XML Editor
Saxonica
Jirka Kosek
...and our long-suffering partners

Sister Conferences

Balisage
The Markup Conference

xmlprague

XML
summer school

Declarative
Amsterdam

Markup UK 2023 Proceedings

by Erik Siegel, Daniel Arthursson, Martin Nilsson, Tony Graham, Sven Reinck, Achim Berndzen, Thorsten Rohm, Martin Kraetke, Daniel Arthursson, Martin Nilsson, Christophe Marchand, Kurt Conrad, Octavian Nadolu, Andrew Sales, Tony Graham, Jorge Sánchez and Liam Quin.

The organisers of Markup UK would like to thank Antenna House for their expert and unstinting help in preparing and formatting the conference proceedings, and their generosity in providing licences to do so.

Antenna House Formatter is based on the W3C Recommendations for XSL-FO and CSS and has long been recognized as the most powerful and proven standards based formatting software available. It is used worldwide in demanding applications where the need is to format HTML and XML into PDF and print. Today, Antenna House Formatter is used to produce millions of pages daily of technical, financial, user, and a wide variety of other documentation for thousands of customers in over 45 countries.

Markup UK

Markup UK

XProc as a command-line application engine

Erik Siegel, Xatapult

This paper explores the use of XProc 3.0 in developing a set of command-line applications for converting XML documents into various derivatives. XProc is used as the core of the application and it experiments with building the application logic in XProc, including command-line handling and file/directory management. The paper proposes two design patterns that can be widely applied to use XProc effectively in developing command-line applications.

1. Introduction

A recent assignment required me to write a set of command-line applications to convert (many) XML documents into various derivatives, some in XML, some in other formats. XProc, especially the recent 3.0 version, seemed to be a perfect fit for this. Being well acquainted with the language, and with Achim's excellent Morgana XProc processor freely available, I decided to use it for the application's core.

But would it also be possible to build the application logic in XProc? For example things like command line handling, doing things based on command line options, file and directory handling, etc.? And do this well, in a satisfying way? After some experimentation I came up with two "design patterns" that are probably more widely applicable.

This paper and the accompanying presentation does not contain ready-made solutions. Instead it explores the two "design patterns" that proved useful in writing such an application: job tickets and command-line wrappers. There are some data and code examples for inspiration and to help you on your way, but no definitive, ready-to-run stuff.

2. What is XProc?

XProc is an XML based programming language for processing documents in pipelines: chaining conversions and other steps together to achieve the desired results. It has been around in its 1.0 version since 2010. The much more versatile 3.0 version was finalized in 2022.

Some high level characteristics:

- XProc is a *programming language*, expressed in XML, in which you can write *pipelines*.
- An XProc *pipeline* takes data as its input (often XML) and passes this through specialized *steps* to produce end results.
- Steps range from simple ones, like reading and writing data, to more complex stuff like splitting/combining/pruning, transformations with XSLT and XQuery, validations against schemas, etc.

- ◇ Within a pipeline you can do things like working with variables, branching, looping, catch errors, etc. Everything is based on the data flowing through.
- ◇ XProc pipelines are not limited to a linear succession of steps. They can fork and merge.
- ◇ XProc allows you to create custom steps by combining other steps. These custom steps can be used just like any other. Custom steps can be collected into libraries.
- ◇ XProc aids in the housekeeping surrounding the processing, like inspecting directories, reading documents from zip files, writing things to disk, etc
- ◇ There is software that can execute these pipelines, the so-called XProc *processors*.

There is of course much more to say about XProc:

- ◇ The XProc main website [https://xproc.org/index.html] has a page with learning materials [https://xproc.org/learning.html].
- ◇ The xml.com [https://www.xml.com/articles/?tag=xproc] hosts a number of articles about XProc, including a full introduction [https://www.xml.com/articles/2019/11/05/introduction-xproc-30/].

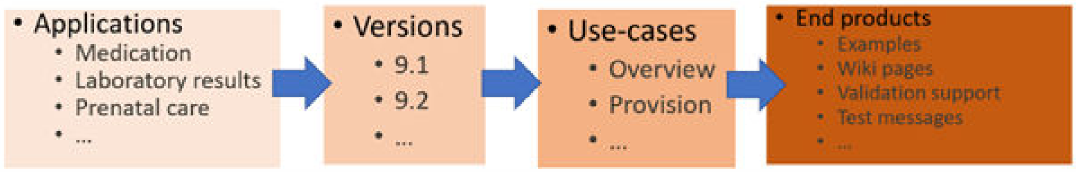
To execute XProc you'll need an XProc processor. Information about this is here [https://xproc.org/processors.html]. The author uses MorganaXProc-IIIse [https://www.xml-project.com/morganaxproc-iiise.html].

3. The problem domain

The application set I'm working on downloads files using a REST service and turns these into various derivatives, some in XML, some in other formats. In doing that it creates sets of files in a specific, complicated, directory structure. There are lots of variants, all doing roughly the same things, but with their own specification, REST URLs, sets of files to copy, and transformation stylesheets.

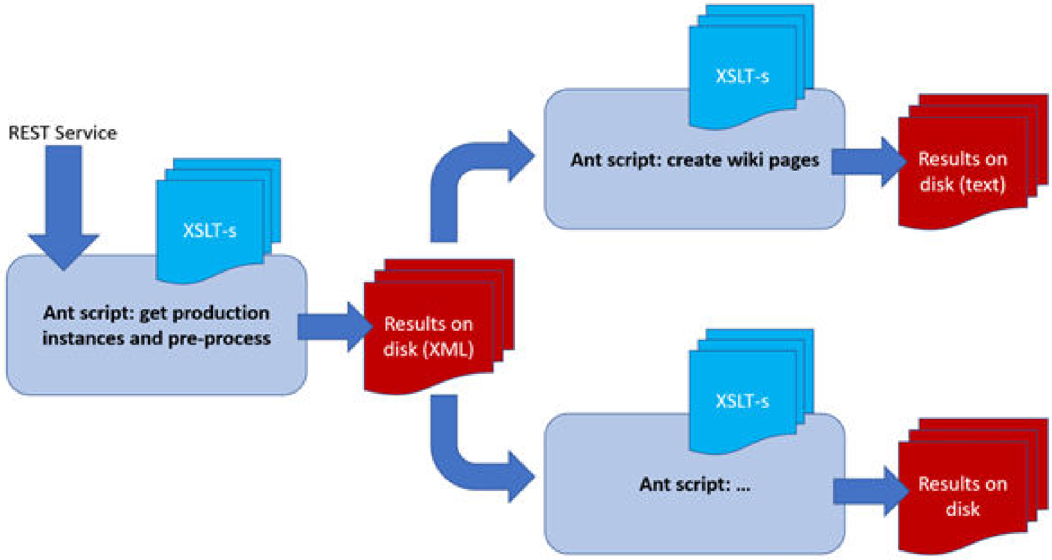
The customer I'm doing this for is Nictiz [https://nictiz.nl/]: the Dutch competence center for digital information management in healthcare. They maintain and support the use of digital healthcare standards in The Netherlands.

There are many domains where healthcare standards are applicable. Each of these domains (called "applications") has their own set of versions, use-cases and end-products. This leads to a combinatorial explosion of stuff that needs to be produced and maintained.



The current implementation is made with Ant as main engine and uses XSLT for the actual transformations. There are over a 100 Ant scripts. The whole set, organically grown over the years, is now considered outdated, hard to maintain and in dire need of refactoring.

Here is an overview of the main data flow in this application. It all starts with specification documents downloaded through a REST service. These are preprocessed (transformed using XSLT) and stored on disk in a complex directory structure. From this, several end-products are made with additional scripting.



4. Is XProc a suitable language?

The first question is of course: does XProc have enough functionality on board to, theoretically, do this? The answer to that is, fortunately, yes:

- REST services can be called using the `<p:http-request>` step (https://spec.xproc.org/master/head/steps/#c.http-request).
- XProc has a set of “file steps” (https://spec.xproc.org/master/head/file/) that allow you to do things like inspecting/creating/deleting directories, write/read/delete all kinds of files, etc
- And, of course, processing documents using XSLT can be done using the `<p:xslt>` step (https://spec.xproc.org/master/head/steps/#c.xslt).

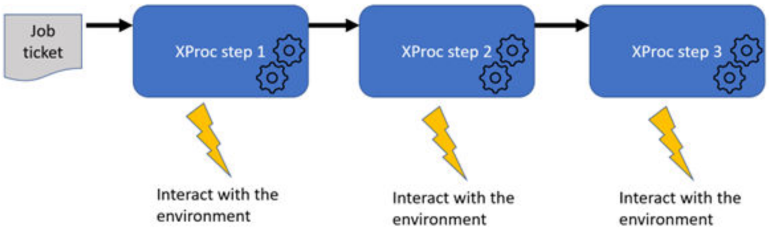
So, from a functional point of view, nothing stands in your way.

5. Design pattern 1: Job tickets

The thing that really helped writing this application well was working with what I call a “job ticket”. A job ticket is some kind of description of what the application has to do: get these files, copy them there, transform these using that stylesheet, store the result there, etc. A job ticket format can be seen as a Domain Specific Language (DSL).

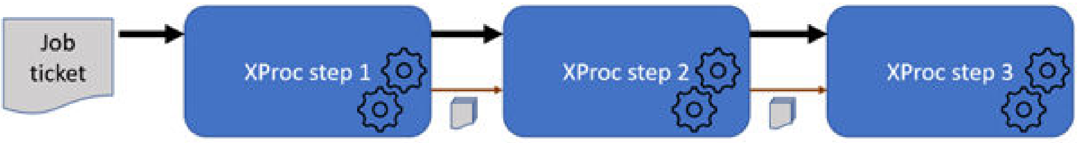
I use XML as the job ticket’s format, but other (XProc supported) formats, like JSON, are of course also possible.

Let’s assume we have such a job ticket (more about this later). The clue to efficiently process this in XProc was in making it the *primary* document that flows through the pipeline:


Job ticket
XProc step 1
XProc step 2
XProc step 3
Interact with the environment
Interact with the environment
Interact with the environment

Usually the documents the pipeline is about are the primary ones, the drivers of the process. But in this case, since we were doing what the job ticket prescribed, it turned out to be way easier to let the job ticket determine the flow. The underlying application logic became simpler and easier to understand.

However, there are of course still the (data) documents that are processed/produced to cater with. The way to handle this is add *another port* on the processing steps and pass them on through this:



In XProc, that’s easy to do, just add another (non-primary) port in the step’s prolog:

```
<p:declare-step … type="steps:step2" name="step2">

  <!-- The job ticket is passed, unchanged -->
  <p:input port="source" primary="true"/>
  <p:output port="result" primary="true" pipe="source@step2">

  <!– Documents are passed using another port (if necessary) -->
  <p:input port="docs-in" primary="false" sequence="true"/>
  <p:output port="docs-out" primary="false" sequence="true"/>


  …

</p:declare-step>
```

◇ The primary output port (called `result`) is defined in such a way (using its `pipe` attribute) that it passes what flows in on the input port (called `source`) unchanged.

◇ The `docs-in` and `docs-out` ports are defined as `sequence="true"`, meaning they can handle a sequence (zero or more) documents flowing through.

And here is an example of how to chain steps like this in an encompassing step:

```
<p:declare-step … name="encompassing-step">

  … (acquire job ticket)

  <steps:step1 name="step1-invocation"/>

  <steps:step2 name="step2-invocation">
    <p:with-input port="docs-in" pipe="docs-out@step-1-invocation"/>
  </steps:step2>

  <steps:step3 name="step3-invocation">
    <p:with-input port="docs-in" pipe="docs-out@step-2-invocation"/>
  </steps:step3>

</p:declare-step>
```

The primary document (the job ticket) “falls through” the steps, because of the implicit primary port connections. The document port connections need to made explicit.

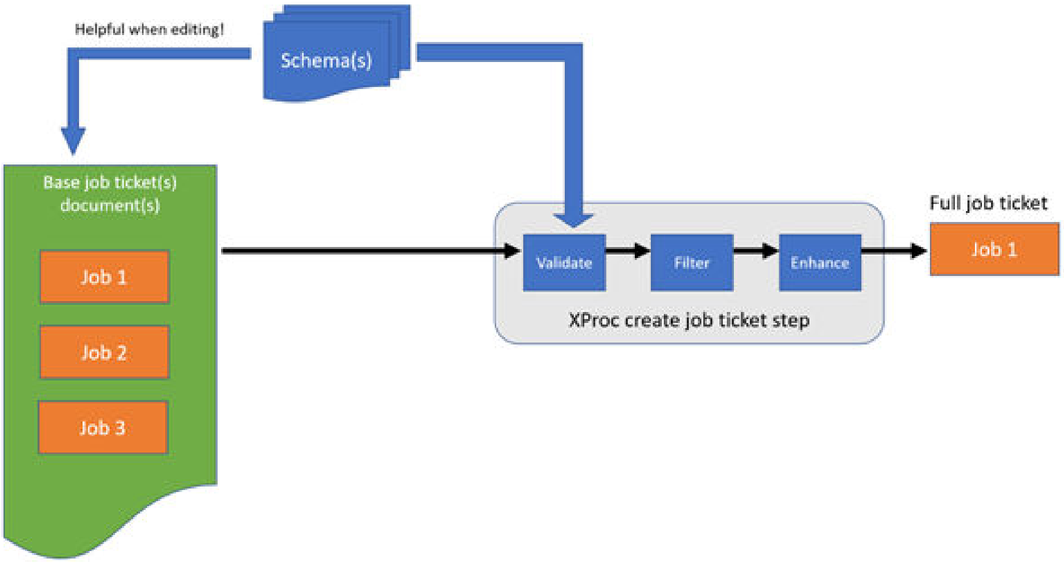
If you need to produce console messages during processing you can use the `message` attribute on steps:

```
<p:file-copy message="* Copying {$hrefTarget} to {$hrefTarget}"/>

<p:xslt message="* Transforming …">
  …
</p:xslt>
```

5.1. The job ticket

Of course, to be able to process a job ticket, we first need to acquire it. Basically this means: find out what needs to be done (for instance by inspecting the command-line arguments) and create it. In my case, it worked like this:



I created an overarching job ticket document that held tickets for several jobs, specified in some internal Domain Specific Language (DSL). Here's a simplified example, just to give you an impression (the details and functionality do not matter here):

```
<jobtickets … >

  <application name="bgz" version="2017" source-project-name="zib2017-nl">
      <setup usecase="kwalificatie" directory-id="bgzkwal">
          <retrieve name="searchset.xml" url="…" directory-id="bgzkwal-instance"/>
          <copy-project-schemas>
            <include glob="*$USECASE.xsd"/>
          </copy-project-schemas>
      </setup>
      <build>
          <stylesheet href="../xsl/…"/>
          <input-document directory="@bgzkwal-instance" name="searchset.xml"/>
          <output directory="@bgzkwal/wiki_instance" name="wiki-bgz.txt"/>
          <parameter name="adaReleaseFilename" value="zib2017bbr-decor.xml"/>
      </build>
  </application>

  <application … >
    …
  </application>
```

```
</jobtickets>
```

For a setup like this I would very strongly advocate the use of XML validation. Actually, my advice is to *always* create a schema for XML documents that are hand-edited, even when it's "just" an internal data file:

- When using a schema-aware editor (like oXygen), it helps you in creating and maintaining the file.
- Consider adding a Schematron schema as well for the more difficult rules (like, for instance, identifier references). It's really helpful when a minor typo in an obscure identifier get's flagged immediately.
- XProc has several validation steps. In fact, validation has become so easy, it's almost criminal not to do it.

A next step would be to filter the main document, based on the job that needs to be done. In my case I threw everything away that didn't belong to the application I was handling. Assume the name of this application is in a variable (or option) `$application`, then the code for this is a single line of XProc:

```
<p:delete match="/*/application[@name ne '{$application}']"/>
```

The next step in processing the job ticket proved *crucial* for the applicability of the job ticket design pattern: job ticket need to be *enhanced*. Enhancement here means: everything you can do or calculate up-front must be done first, before the actual processing starts.

When file or directory names are involved, calculate their full absolute values up-front. For instance, in my case, names sometimes depended on global parameters (some main directory), attribute values of the current and parent elements (subdirectories) and some global settings made elsewhere (version numbers, etc.). All this was tricky to compute. Also: absolute file and directory names *must* be URIs (having `file:///` in front). You have to make sure this is the case before using them in XProc steps.

XProc support the full XPath 3.1 language, so, yes, in theory, you could do these kinds of computations directly in your XProc pipeline. However, this would lead to very complicated and overly long XPath expressions, especially because XPath functions (which would help, a bit) are not (yet?) supported. I would recommend doing all these up-front enhancements in XSLT, which is a language much more suited to these kinds of things.

Here is an example of a snippet of an enhanced job ticket:

```
<application _target-dir="file:///C:/.../lab/3.0.0"
             _source-dir="file:///C:/.../lab/3.0.0"
             name="lab"
             version="3.0.0">

  <setup _target-dir="file:///C:/.../lab/3.0.0/sturen_laboratoriumresultaten"
         _source-dir="file:///C:/.../lab/3.0.0/sturen_laboratoriumresultaten"
         usecase="sturen_laboratoriumresultaten"
         directory-id="slr">

     <copy-data _target-dir="file:///C:/.../ada_instance_repo"
                _source-dir="file:///C:/.../sturen_laboratoriumresultaten/ada_instance_repo"
                target-subdir="ada_instance_repo"
                source-subdir="ada_instance_repo"
                directory-id="ada-instance-repo">

        <include _pattern="\.xml$"
                 glob="*.xml"/>
```

```
    </copy-data>
```

Target and source directories are computed at every level of the document (the `__target-dir` and `_source-dir` attributes). When the job ticket processing wants to do something, based on one of these elements, it can simply lift the URI from the applicable attribute.

Another enhancement here is that a UNIX-style glob was provided (the `include/@glob` attribute). However, XProc works with regular expressions with regards to including/excluding files, so this glob had to be converted into an XPath regular expression. This was added in the `_pattern` attribute.

One last thing about these kinds of data documents is that XProc makes it very easy to setup support for includes. Create an include structure using XInclude `<xi:include href="…"/>` elements (the namespace prefix `xi` is bound to the XInclude namespace `http://www.w3.org/2001/XInclude`). In XProc, the `<p:xinclude/>` step does all the work for you and flattens the document by resolving all the includes. That's all there is to it.

When processing the resulting job ticket, we need take decisions about what to do, in my case based on element names. In XSLT this would be rather simple: call `<xsl:apply-templates>` and write a template for all applicable elements. However, XProc does not have such a mechanism. You'll, unfortunately, have to write dispatching code like this:

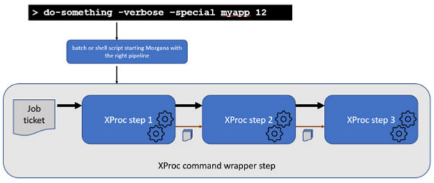
```
<p:for-each>
  <p:with-input select="/*/*"/>

  <p:choose>
    <p:when test="/*/self::copy-data">
      …
    </p:when>
    <p:when test="/*/self::copy-schemas">
      …
    </p:when>
    <p:when test="/*/self::build">
      …
    </p:when>
    <p:otherwise>
      … (error)
    </p:otherwise>
  <p:choose>

</p:for-each>
```

6. Design pattern 2: Command line wrappers

So assume we have created a pipeline using the job ticket pattern, how do we get this started from the command line? It turned out that adding a "command wrapper step" made a lot of sense. Here is how this works:



```
> do-something -verbose -special myapp 12
```

batch or shell script starting Morgana with the right pipeline
Job ticket
XProc step 1
XProc step 2
XProc step 3
XProc command wrapper step

- The command on the command-line starts a batch/shell script that fires the XProc processor (with all the correct flags and arguments).

- This starts a “command wrapper step”: an XProc step whose sole purpose is to unravel and interpret the command line.

- The command wrapper step then fires the appropriate step(s) that implements the actual functionality of what we’re trying to achieve.

Why is this useful? In XProc steps you’ve written can be easily re-used. So it makes sense to keep steps that implement application logic “pure”: as unaware of the environment they’re running in as possible. Sometimes you’re using them as part of a command line tool and, maybe, sometimes in other contexts. Making them aware of things like command line arguments severely hinders re-use. So best to keep the interface of the functional steps simple using ”straight” options (booleans, strings, numbers etc.).

What would such a command-line wrapper step look like? You can of course make this is complex as you like. I used a fairly simple setup that turned out to be sufficient for my purposes:

```
<p:declare-step … >

  <p:option name="commandLine" as="xs:string"/>

  …

  <p:variable name="commandParts" as="xs:string*" select="tokenize($commandLine, '\s+')[.]"/>
  <p:variable name="commandFlags" as="xs:string*" select="$commandParts[starts-with(., '-')]"/>
  <p:variable name="commandArguments" as="xs:string*" select="$commandParts[not(starts-with(., '-'))]"/>

  <p:choose>
    <p:when test="'-help' = $commandArguments">
      … (output a help text)
    </p:when

    …

  </p:choose>

</p:declare-step>
```

- The full, unparsed, command line is passed to the command wrapper step in the `$commandLine` option.

- This is tokenized into words (based on whitespace as separator) and then separated into:

  - Flags (anything that starts with a hyphen, for instance `-help`)

  - Arguments (anything else)

- A `<p:choose>` makes decisions based on the flags/arguments present.

7. Wrap-up and conclusions

XProc can be used very well for implementing command line applications that work lots of documents, processing/building directory structures, etc. There are two design patterns that help:

- Job tickets, using some internal Domain Specific Language (DSL), greatly simplifies the application logic.

- If you handle the command line in a separate wrapper step, the steps that implement the actual functionality become more reusable.

Working with XML inside a web browser

Edge computing for XML

Daniel Arthursson, CloudBackend.com

Martin Nilsson, xios3.com

Decentralization of data and applications with edge computing is coming outside the XML community. Can we revitalize web browsers and shift XML processing from the server side to the client side with added benefits of improved speed and low latency? Can we make XML an alternative to JavaScript and JSON development within web browsers? This requires client-side database technologies, client-side XML programming languages, support for XSLT, and powerful ways to address standalone XML documents as well as XML stored in document databases or XML supplied through cloud-based API calls. This calls for data abstraction and a unified data model within the web browser. This paper talks about two large projects aimed at solving the client-side equation of XML and putting XML on the map for web development again.

1. Introduction

Everything started out great for XML with growing support in the early 2000s, but since 2010 and forward we have seen a massive decline in client-side XML support with known bugs in XSLT in both Firefox and the WebKit project (Chrome, Edge, Safari, Opera); going unfixed. We know, as we have reported issues and provided C++ patches for XSLT bugs in both browser platforms, but to our disappointment seen them linger or be dismissed.

XPath is stuck on version 1.0 in web browsers, and promising efforts to reimplement XPath in JavaScript by Google have been abandoned since many years. Here we have once again stepped in and fixed many bugs and made the Google project run with the latest build technologies. This time we consider taking over management of the open-source project.

We have also seen XML staple technologies like XML Web Services become replaced by REST APIs, and XML as a data transport format being replaced by JSON and GraphQL. It is easy to see no hope for XML in web technologies; while it is still extensively used in enterprise solutions. However, good examples like Frameless XPath 2.0 and XSLT 2.0, and recently the Saxonica port for JavaScript.

This has generally pushed the use of XML to the cloud and the server side, where the latest version of XSLT, XPath, and XQuery can be used. The browser has mostly been used as a dumb rendering engine for HTML and JavaScript generated in the cloud.

With improved JavaScript engines and execution performance, there are finally ways to circumvent the obsolete XML technologies within the browsers. New JavaScript implementations running within the browser sandbox are in many cases more performant than decades-old C++ implementations of the same technologies. There is thus ample opportunity to extend the native built-in web browser XML technologies with new technologies for building rich web applications, forms, XML generating applications, and caching XML data within the browser for local client-side processing. Technologies that can bring XML back to its web glory days and have the potential to outperform the latest React, Angular, jQuery, JSON, and GraphQL technologies.

Employing the browser for client-side processing, rendering, execution of XML logic, and caching of XML data, will enable device edge computing for XML. Edge computing is the decentralization of cloud computing, to have logic and data not in a single place, but available in many places. It promises low latency and new XML-based ways of writing web applications.

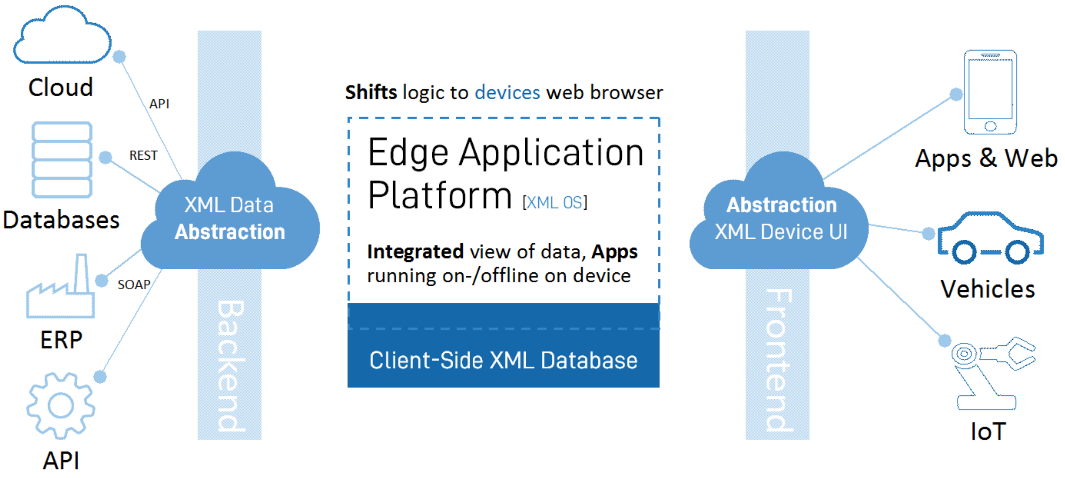
This paper will discuss technologies employed by two large projects that together are aimed at super-powering web browsers on smartphones as well as laptops for the use of XML. Both projects aim for general XML use, but in particular for building rich web applications that consume and keep data in sync as XML across the wire as well as within the software application.

2. Project Goals

First, let us look at what ideas started the development of the two projects. There were a couple of clearly defined goals:

- ◇ Shift server-side execution to client-side, typically the web browser. This is sometimes referred to as device edge computing.
- ◇ Shift all HTML generation client-side, i.e., no user interfaces generated server-side.
- ◇ Shift all program logic client-side to enable offline-first applications that continue to work while not connected to the Internet
- ◇ Provide a high-level XML abstraction of describing user interfaces instead of programming the user interface with JavaScript and HTML5.
- ◇ Provide an event-based programming language in XML as a higher abstraction to write logic and orchestration in addition to JavaScript.
- ◇ Implement a full separation of user interface (view), program logic (controller), and model (data) according to a pure MVC design pattern.
- ◇ Intelligent data binding of XML to UI components.
- ◇ Provide data abstraction towards all types of data sources; client-side, XML Web Services, REST, and JSON.
- ◇ Use the cloud backend as a pure data repository accessed through an API that provides data, authentication, and security.
- ◇ Enable anyone to add support for any XML application (language) and allow the platform to recognize and act when encountering XML applications.
- ◇ Reuse as much as possible of the browser capabilities, extend or replace when necessary.

Figure 1. Abstraction based on XML in all areas is key.


Cloud
API
REST
Databases
SOAP
ERP
API
XML Data Abstraction
Backend
Shifts logic to devices web browser
Edge Application Platform [XML OS]
Integrated view of data, Apps running on-/offline on device
Client-Side XML Database
Abstraction XML Device UI
Frontend
Apps & Web
Vehicles
IoT

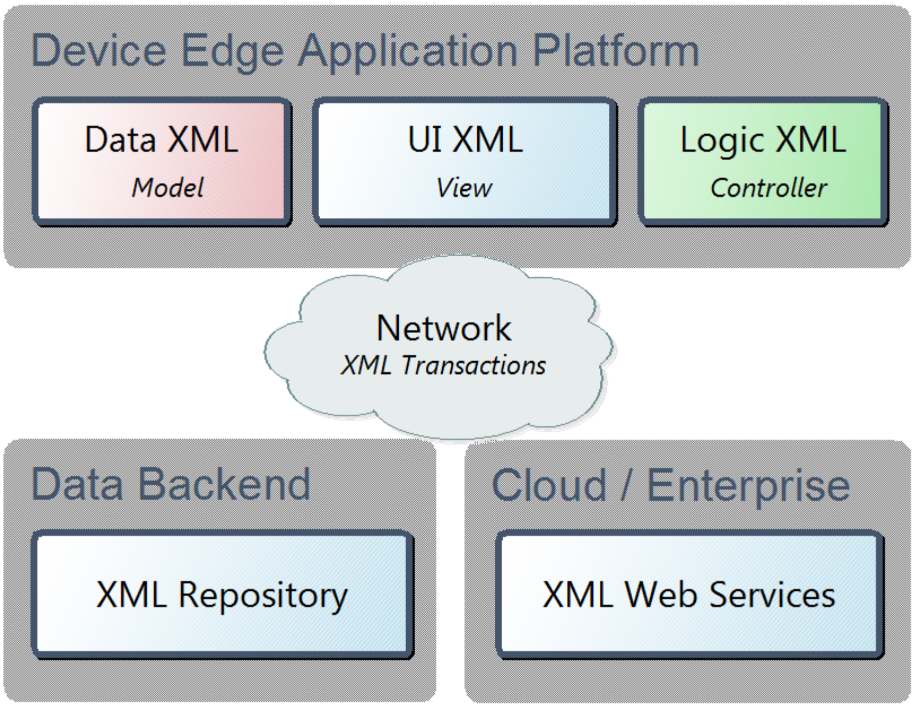
3. What new XML technologies are required?

The set goals, including the full separation of Model, View, and Controller of an application and the full utilization of XML, requires three new XML applications to be developed:

- A generic data model in XML that can hold any XML application needed by the developer defined software (model).
- A way to describe user interfaces in XML (view).
- The ability to write program logic for the application using XML (controller)

All with the ability to exchange data with a cloud-based backend using XML and XML delta transactions.

Figure 2. XML applications required to be developed



The application written on top of the Device Edge Application Platform also requires some type of XML application as a manifest to be described and linked to all its required resources. The above figure 2 suggests a technology that consumes and produces XML, that runs applications written in XML, and communicates changes to XML using XML delta transactions.

In addition, a specialized XML application to allow declarative definitions of intelligent data bindings between XML data and user interface components described in the UI XML language is needed. This will allow any developed UI component like a tree, menu, tab strip, or toolbar to be bound to any type of XML and listen to data changes on the XML or update the XML according to user interaction.

It would be wise to reuse as much as possible of existing XML standards like XSLT, XPath, XPointer, XLink, and ATOM to be used together with any new XML applications developed for the projects.

4. Technical Challenges

The first obvious question is how this should be developed inside a web browser using JavaScript and HTML5. Should an existing JavaScript framework be used? If so, are they compatible with being controlled from XML and working with XML, or do we need to create an entirely new technology stack specialized on XML?

XForms was not a technology we saw could be used for building a new presentation software, calendar, or email software. We needed something more expressive that could replace the role of traditional software development.

This project thereby came to the conclusion; we needed something that was thinking, breathing, and communicating XML even in its sleep. We also needed several engines that could interpret and execute the new XML languages for applications, UI, logic, data bindings, data model, and data types – i.e. something that could make some sense of all the new markup languages we needed to come up with.

We also needed the ability to in the future be able to extend the vocabulary of the user interface XML language with new markup representing UI components, either by XSLT or by JavaScript.

Another property was that it should be extensible by any new XML application and storage format designed by any software application developer using the platform. The platform needed to be able to understand these new XML applications, to act on them, to make sense of them, and to be able to manage the data for them. In a sense making the entire Device Edge Application Platform an XML editor in the form of a fully user-defined application UI for any possible XML application (language).

The custom user-defined software applications developed in XML we imagined to be built on the platform were as advanced as; presentation editor, word processor, calendar, contacts, bookkeeping, document management, database, instant messenger, photo album, music player, and similar.

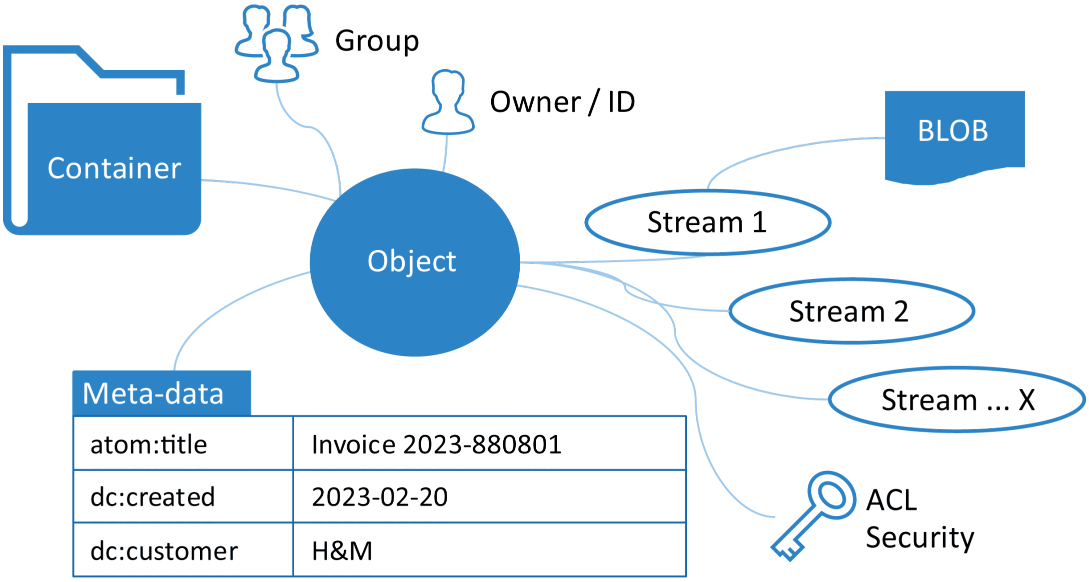
Three technical challenges stood out; 1) how to make any UI component understand any data model it could be bound to and how the UI component should then be able to know how to extract the right data, 2) how to get the browsers built-in parsing of XML Document Object Model (DOM) and XPath to be able to handle advanced XML documents with multiple namespaces while keeping the client-side cached data models as DOM objects to minimize parsing of XML, 3) how to keep parsed XML DOM objects up to date with what is happening in the cloud and to keep any user interface components up to date to any changes in the XML DOMs it is intelligently bound to.

5. The XML data model layer

If everything from enterprise to productivity applications should be possible to build on the platform, a versatile data model is required. As most XML writers are used to create XML documents, for instance an XML document describing an invoice, a calendar event, or a contact, these XML document represent repeating collections of data, like multiple calendar event documents. The data model thereby needs to be able to manage repeating XML documents – collections of documents.

The data model layer also needs to be able to extract meta-data that should be searchable to not require applications to open every single XML document and parse them to find what to display or search for. This means that the data model needs some sort of data-type definitions that can be associated with any XML application and defines how to apply meta-data extraction, what to do if such a document is opened – or any appropriate action on the document. The data types can also allow other meta-data to an XML application to be added like for instance a default icon, namespace definition, file extension, and mime type.

Figure 3. XML-based data model abstraction using containers and objects



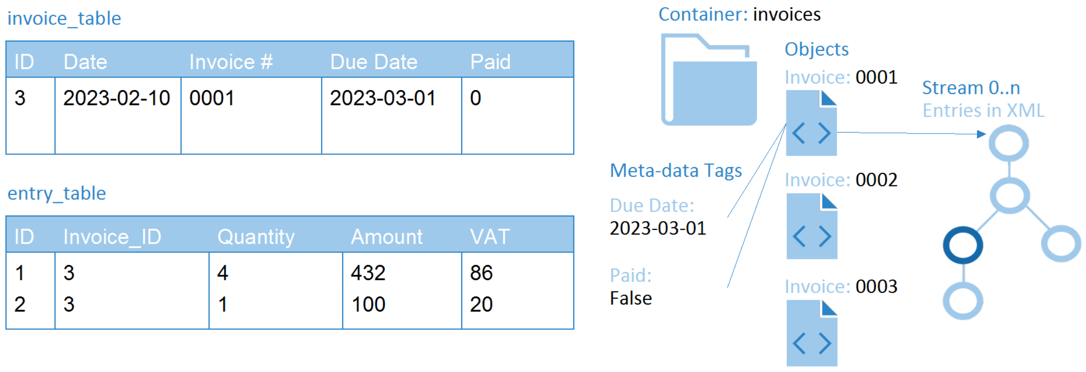
atom:title	Invoice 2023-880801
dc:created	2023-02-20
dc:customer	H&M

We came up with a data model based on containers and objects, where containers can be hierarchical and objects can have zero to many streams associated with them, ranging from binary objects to XML documents. In its simplest form, a container is equivalent to a folder in a normal file system, and an object with one stream in the form of an XML document would be like a normal XML document stored in a file system. Objects can then also have additional metadata like key/value data associated with it to be searchable, either external file data like the name, creation date, filename, and access rights, or extracted data from within the document itself. This makes it easy to “attach” binary file formats to a XML document, as the binary additions are additional streams to the XML document object.

Querying a container returns an answer in the form of an Atom XML document according to the Atom Syndication Format by the IETF AtomPub Working Group. This format is extended using our own namespace to add capabilities to return containers in addition to objects described by atom:entry elements.

When creating software applications there is often the need to work with relational databases as data sources or to move data from relational databases into the data model. This can be done by mapping tables to containers, with the added benefit of getting hierarchical containers, and rows to objects. Columns are achieved with key/value meta-data, and complex column fields are done by storing entire XML documents as streams to the object. Binary BLOB fields can also be stored as streams.

Figure 4. Mapping relational data to the XML data model



invoice_table

ID	Date	Invoice #	Due Date	Paid
3	2023-02-10	0001	2023-03-01	0

entry_table

ID	Invoice_ID	Quantity	Amount	VAT
1	3	4	432	86
2	3	1	100	20

The data model may be cached in-memory of the browser or persisted to disk using the local store technology of web browsers. Any changes to the data model are done through the platform abstractions and not directly on the XML DOM objects. This allows the browser-based transaction manager to coordinate changes to the user interface as well as to the cloud where a transaction coordinator is located that coordinates transactions across users and connected devices. It also allows applications to continue to work with the data model if the network temporarily goes down or if the application is used in an offline scenario. If the browser is closed, any uncommitted transactions may be first written to disk.

Any application working with the data model automatically becomes capable of collaboration and simultaneous editing across multiple connected web browsers (single user with multiple screens or across several users), as any data changes automatically are coordinated through XML delta-changes while the data model keeps track of all identities simultaneously working with the same XML document.

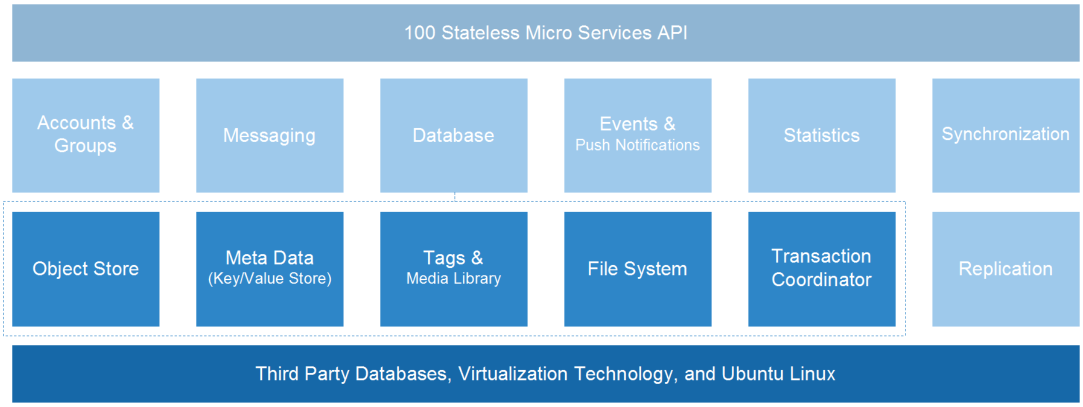
This adds another fundamental perspective to the data model. It is aware of the identities of authenticated users and can keep security based on Access Control Lists (ACL), roles, and groups across containers and objects. It also opens up the possibility to share containers and/or objects between identities and/or groups.

The data model's cloud-based counterpart can be any implementation adhering to the same XML-based communication channel API on the Device Edge Application Platform or it can be any existing web API using SOAP, REST, MQTT, GraphQL etc. adapted to the internal data model through client-side protocol adapters.

From an application's point of view, the entire data model is simply a set of file systems with XML documents, all accessed through Unique Resource Identifiers (URIs). This abstracts where data is located, how it is updated and saved, and what the underlying representation is. Every single piece of data can be accessed through a URI and an XPath. The drawback of course that for all non-XML data sources the data has to be transformed through a bijective function which increases overhead and can cause a trade-off between data loss and the usefulness of the XML representation from the bijective function. This is however in real life not an issue, as most data that is not in XML format is in JSON or GraphQL, which only provides a sub-set of the descriptive powers of XML.

To determine the set of required server-side capabilities and validate APIs a large amount of client applications representing the full suite of productivity and social applications of a typical desktop computer were developed. This resulted in around 100 server APIs that together implemented support for the data model. They are covering user management, communication, data storage, collaboration, messaging, and more.

Figure 5. Cloud-based support in the form of an XML repository serving the data model


100 Stateless Micro Services API
Accounts & Groups
Messaging
Database
Events & Push Notifications
Statistics
Synchronization
Object Store
Meta Data (Key/Value Store)
Tags & Media Library
File System
Transaction Coordinator
Replication
Third Party Databases, Virtualization Technology, and Ubuntu Linux

6. The XML user interface layer

There is a concept within software development known as “design patterns” where you identify specific patterns in software design and generalize them for reuse. Several such patterns have been proposed and are in common use. One well-known design pattern is called “Model-View-Controller” (MVC) where you separate the code responsible for the underlying data model, the code that presents data for the end user, and the code that modifies the data. The idea is that by keeping these parts separate it is possible to change any of these aspects without altering any of the other code. The intent when designing the XML application for the user interface language was to keep it at a high abstraction level without mixing in any programming code, to maintain a pure MVC separation, and without too many cues about how it should be rendered. This allows the user interface to be rendered into the device form factor, each device’s expected behavior, and allows for effective use of themes for design and style.

XML has proven itself to be a popular description language for representing application UI layouts. Notable examples are the layout language for Glade Interface Designer for GTK, the layout language for Android, and the XAML language for Windows Presentation Foundation from Microsoft.

Common between all these languages is the same principle that worked so well for HTML, to have nodes representing user-accessible components e.g., buttons, in a tree of layout constructs to position the component on screen or in a window. In the case of Glade/ GTK, they use a small abstract language to allow for a minimal XML Schema while the other two name individual components as tags to make the language more accessible but extensibility harder. We can give similar examples from popular web frameworks like Vue and Angular, but there the line between declarative layout language and code is not as strict, which is where a lot of languages fail with respect to the Model View Control design pattern.

The idea with this project’s UI XML language is that it should be possible to create almost like a wireframe version of the UI in an XML document, then build the entire working application and at the same time have someone working on another UI XML document that makes the application look beautiful. When the designer is done, the files should be possible to swap out and the application would get a facelift.

A second dimension to this is that it should be possible to create visual themes for all the user interface components and windows, which then by the change of an attribute value of the application manifest should be possible to change and then affect the entire rendering of the application without changing any code.

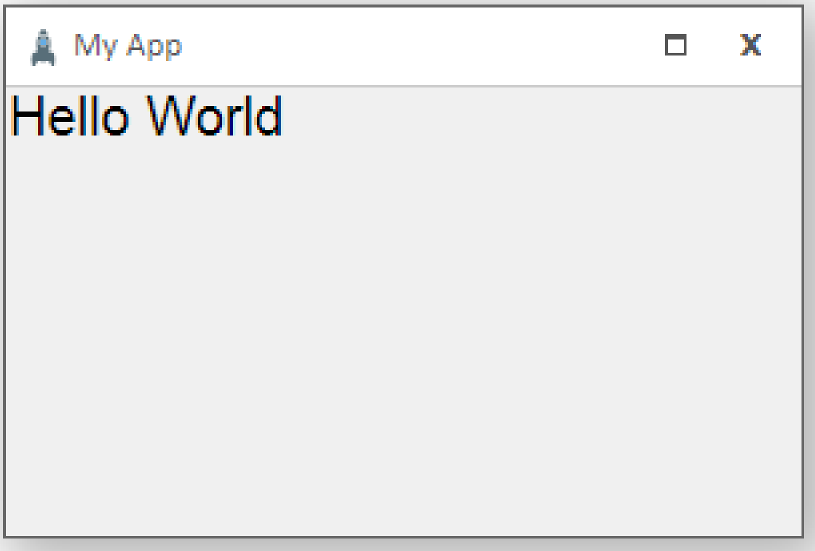
The markup for the UI XML should also be as easy for someone to learn as using XHTML, this requires it to use a single implicit namespace with the addition of more namespaces only for advanced users. This creates some issues we have not fully resolved yet. The system is built so it allows for dynamic updates of the set of UI components, which in turn means the XML schema would have to be dynamically generated for anyone using this feature or the use of additional namespaces to add user developed UI components. This issue is the same as XHTML has with web components extending the language.

When creating software, there is also often a need to support multiple human languages of an application. Including all translations into the UI XML would make it hard to read and would go against the goal of separation of concerns with MVC. Instead we need an elegant mechanism to tie in translations for the currently chosen language, stored in a separate XML application for translations. The fairly non-intrusive and easy-to-use solution is using a localization namespace, using, for example, the prefix of “l”, and if then the attribute value for “title” should be translated, writing it as “l:title” instead. A transformation pass before the document is interpreted replaces all attributes in the language namespace with corresponding attributes in the default namespace, but with translated values. While it is natural to place localized data in text nodes, a drawback with this approach is that all localized data has to be in attribute values as text nodes cannot be prefixed to a namespace.

The following is a simple hello world application written in the UI XML language where the view element represents the outer boundary of the application, typically a screen or a window, the panel represents a layout container, and the button element the component to be placed inside the panel.

```
<view name="myView" title="My View" icon="icon://rocket">
  <panel>
    <button name="myButton" text="Hello World"/>
  </panel>
</view>
```

Figure 6. Rendering of the application view



The coupling of any data model or event-driven actions to the user interface is done entirely outside the UI XML in declarative XML, which means it is possible to reuse the same UI XML and even most of the program logic of an application although they work with completely different XML application documents (data models and/or data sources).

This is achieved by having every UI component like a tree, button, menu, toolbar, etc. bound directly to a fragment of an XML document where the fragment is pointed out using XPath. Events and data changes across UI components, the transaction manager, delta transactions, and the data model are all done using XML and XPath. This makes the entire event and data update bus of the Device Edge Application Server driven by XML. How the UI components should interpret the XML it is bound to are defined by the declarative data bindings. This means that UI components developed for the platform are generic and not specific or tied to any XML application, they can be reused by a software developer across all projects, as they in a simple way are taught how to understand the XML application, its hierarchy, elements, and attributes.

The following is an example of a tree UI component with rules describing how it should interpret data that it is bound to. The tree UI component is then able to render the hierarchical nodes of the tree out of the XML data model by itself. Rules can also be described in the declarative data binding for better reusability outside of the UI XML.

```
<view name="treeView" title="Tree Demo" icon="icon://pieces" width="250">
  <panel width="200" type="row">
    <tree name="myTree" width="100%" height="100%" icon="icon://rocket">
      <rule match="components" display="'Component List'" icon="icon://pieces" open="true"/>
      <rule match="component" display="@name" icon="icon://piece"/>
      <rule match="component/documentation" display="description" icon="icon://pencil"/>
    </tree>
  </panel>
</view>
```

Anyone familiar with templates in XSLT can see the similarities to how rules match underlying elements in the data model using XPath.

The final example is of the iconlist UI component that in this example is bound to the result from a container query, an Atom XML document extended to support the containers, symbolized by fs:folder elements in the feed.

```
<iconlist name="lists" width="100%" height="70" layout="sectionlist"
  scroll="true" deselect="false" iconsize="32">
  <rule match="fs:folder">
    <item text="{substring-after(@name, ' ')}" icon="icon://check"/>
  </rule>
</iconlist>
```

The rule will make sure that the iconlist component is fed one item for every fs:folder (container) that the query results in. The iconlist component is bound to the root element (atom:feed) of the Atom XML document, which means that all XPaths inside the rule is relative to this XML fragment. This is similar to how matched xsl:templates changes the current context for sub-sequent XPath expressions within the template. XPath functions are available to be used within the text attribute value, similar to how they would be used in XSLT.

7. The XML program logic layer

Less common than representing layout with XML is to represent processing steps with XML, but there are again plenty of examples. Most notable is of course XSLT with its XSL document, execution environment of constants, and input document which together generate a third output document. In XSLT you define template rules, some of which are triggered directly by the nodes in the input data and some which are called like functions from other templates. Inside the templates are imperative statements that are executed in sequence, sometimes branching on conditionals, or iterating over a set of element nodes.

Apache ANT, used as a build system, most notably for Java projects, in a similar way has an execution environment of constants but instead of traversing an XML tree in the input document to output resulting markup into a document it traverses a dependency tree and creates resources in a filesystem. Just as with XSLT there is a list of rules, called targets, that are executed depending on the input statement for the execution and the state of files it is processing, e.g. has the file been modified since the last time we ran the build script. Inside the targets, the individual processing steps are then executed in order.

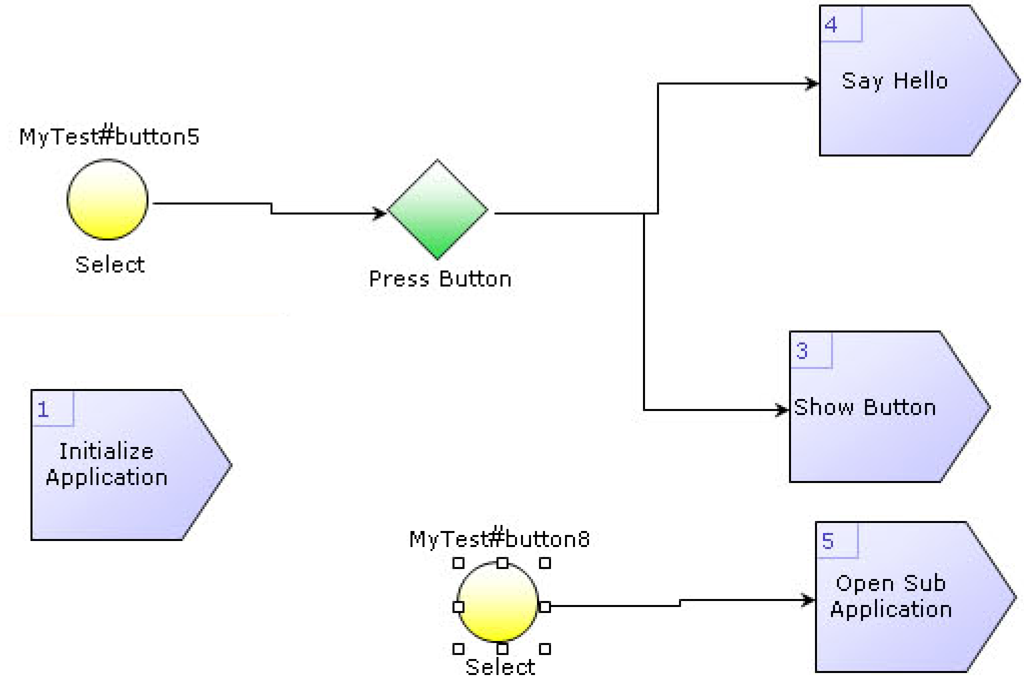
Both XSLT and ANT are non-interactive and non-responsive in the sense that it represents a contained work package that is started at one point after which it continues its set of instructions until it is finished or encounters an error. To build interactive applications it must be possible to have the user interactions trigger execution and allow execution steps to be concurrent with each other. This can be solved by creating an event system where user interactions such as clicking a button or pressing a key generates events which in turn trigger execution steps in the XML code. The logic XML enables event listeners to be bound to different execution steps

We decided on creating a language based on steps containing logical operations performing actions, triggered by events sent from the system, components, and changed documents on the Device Edge Application Platform event bus. Since the platform supports multiple running instances of a single application, such as an email program having several emails open in multiple windows, or a text editor with multiple documents open, and that we would not like the browser memory to have multiple copies of run-time objects and XML DOM objects we designed it so the same logic XML can serve all concurrently running instances of the application.

The idea of the XML logic language is to work as a glue, binding documents from the data model to one or more user interfaces (views), talking to external cloud APIs to integrate data, and to process user-generated events and interactions. It should also be able to manage inter-application interaction when several software applications are running side by side in the Device Edge Application Platform.

Another design consideration was that it should be possible to create a visual editor that allows developers to use drag-and-drop to build the logic XML and see it as a visual process drawing. In the following implementation, yellow circles are triggers, green represents decisions and the rest are process steps. The notation of the names above the yellow triggers is like a XPointer style referencing mechanism that allow referencing of a named view and UI component from the logic XML.

Figure 7. Visualization of the logic XML language created inside a web-browser



Since modern-day web applications are working asynchronously to not tie up the user interface while waiting on a server request to complete, the logic XML programming language needs to make sure everything underneath it runs asynchronously. It gives the developer the illusion that the programming can be done synchronously – significantly simplifying the developing experience – while in reality it runs asynchronously and possibly concurrently.

This is a small example that makes the button in the previous view example close the view.

```
<process name="myProcess">
  <trigger view="myView" component="myButton" event="Select"
    step="closeStep"/>

  <step id="closeStep">
    <operation name="close" value="myView"/>
  </step>
</process>
```

The following is a complete example of an application manifest that includes both a view and a logic process, they can both be linked into the manifest using the W3C XLink standard instead.

```
<application name="My App" icon="icon://rocket" instances="0" theme="marble">
  <resources>
    <item name="employee">
      <card>
        <name>Gandalf the White</name>
        <occupation>Wizard</occupation>
      </card>
    </item>
  </resources>
  <view name="App" title="My App" icon="icon://rocket">
    <panel>
      <label size="20" name="myLabel" default="This is a label"/>
    </panel>
  </view>

  <process name="My App - Process">
    <trigger view="App" event="Loaded" step="init"/>
    <step id="init">
      <operation name="bind" value="#employee">
        <component view="App" name="myLabel" select="/card/name"/>
      </operation>
    </step>
  </process>
</application>
```

As soon as the application manifest has loaded the application, including opening its view, the trigger will listen to the “Loaded” event and call step “init”. The init step will use the process operation called “bind” to bind the temporarily created XML document added to the data model defined above under resources to the label component. The label component will thereby change its content from “This is a label” to “Gandalf the White”. If any other code or external application alters the data model, the label component will automatically update its text to the new state of the data model.

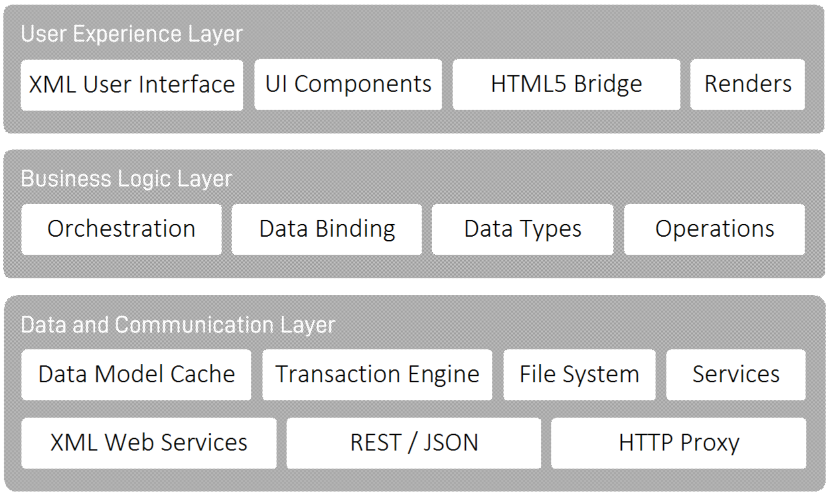
This means that the logic XML language together with the intelligent data bindings removes all need to call server APIs to get data changes, then update a local object data model in memory, then listen to changes to that object model and use set and get methods on the UI component to update it programmatically and vice versa. All of this is done by the Device Edge Application Platform.

8. The resulting architecture of the XML edge platform

The following architecture overview shows the major components of the XML Device Edge Application Platform. Obviously, there are many more components to make it work, but the following are the core functionality.

At the user experience layer, the application manifest, the UI XML, but also XSLT renderers and JavaScript code that create the XHTML output are located. This layer can be extended using the component API by any developer.

Figure 8. Architecture of the XML Device Edge Application Platform



The business logic layer consists of modules to manage the XML process logic markup as well as the intelligent declarative data bindings, the data types for making the platform aware of any XML applications, and the API to extend the logic XML with arbitrary program logic operations.

Finally, the data and communications layer provides support for the XML-based data model. It allows channel services to be built that interface into any backend system, API, or database. It provides the container/object file system abstraction used to consume the data model and also includes ready-to-be-used communication channels for speaking with XML Web Services using SOAP, REST APIs, interfacing with JSON or any XHTML compliant website using the HTTP proxy.

9. The two projects

Everything discussed in this paper that has to do with the Device Edge Application Platform is the work of XIOS/3 AB (https://xios3.com), while everything that has to do with the data model and Singularity Database is the technology from CloudBackend AB (https://cloudbackend.com). These two projects are not yet launched and available commercially, but we expect them being available within the next couple of months.

Several concepts and ideas discussed in this paper are patented and protected by the following issued US patents (18 patents in total). Further information about the XML Internet Operating System (XIOS) can be found in the patent descriptions in the link below.

Table 1.

US8239511	Network operating system application packaging
US8843942	Interpreting semantic application code
US8959123	User interface framework
US8099671	Network operating system opening an application view
US8954526	Network operating system data source abstraction
US8112460	Framework for applying rules
US8996459	Offline and/or client-side execution of a network application
US9344497	State management of applications and data
US8615531	Programmatic data manipulation
US8280925	Resolution of multi-instance application execution
US7917584	Gesture-based Collaboration
US8108426	Application and file system hosting framework
US8156146	Network file system
US8234315	Data source abstraction system and method
US8620863	Message passing in a collaborative environment
US8688627	Transaction propagation in a networking environment
US8738567	Network file system with enhanced collaboration
US9071623	Real-time data sharing

10. Future development for the two projects

At the moment the two most frustrating problems with using only the built-in XSLT and XPath engine of web browsers, even though we have a lot of custom extensions, is the inability to support additional XPath 2.0 and 3.1 functionality and to have our own extension functions. The same goes for being able to add extension functions to XSLT. Having that capability would allow much tighter integration between the Device Edge Application Platform and the XML world.

It is also very frustrating sometimes to not be able to debug XSLT and connect the ability to step through XSLT with the ability to step through the XML process logic using the built-in debugger we created for XIOS/3.

We have thus written a new very nimble XML parser in JavaScript, a new XPath engine is under development, and we then aim to write our own XSLT implementation in JavaScript. The reason of doing all this work is because of the tight integration that we believe we need and being able to optimize the code as we heavily process and use XML. There will also be C++ versions of the XML, XPath, and XSLT engine available for inclusion in the Software Development Kits (SDK) offered by CloudBackend that allow integration into native applications in Swift (iOS), Java (including Android), and C++ (Windows, Mac, Linux, and IoT) that tap into the data model and XML applications that run inside the browser. We want the same XPath and XSLT to be able to run in the web browser, the cloud, and on mobile apps for cross-device development. This will probably be material for another future paper.

11. Conclusion

It is possible to create quite a sophisticated XML development environment with edge device acceleration and caching already today. Managing to build the entire infrastructure is a lot of work and includes building a new JavaScript framework and many more technologies not normally associated with running within a browser like a transaction engine and multi-instance support of applications within a single browser tab.

The end result is amazing from an XML perspective. We can get an entire environment where everything is XML, from the data format we open and store, to the applications designed to work with the data, all the way to events created because of a user clicking on something or data being update in the cloud.

A lot of technical challenges are solved by using XML as the common denominator throughout the entire software project. The extensible nature of XML with the ability to add new elements later in a project or mix and merge several XML applications into one XML document using namespaces are very powerful tools available to a developer.

The edge computing aspect of things, running most of the application logic inside the browser or even the entire application logic, brings some interesting effects to the table like:

- Changing the role of the cloud to become a storage and coordination point, utilizing all connected clients for most application logic processing, dramatically reducing the cost of using cloud computing since fewer server instances are needed.

- Reduced required network bandwidth as clients cache data locally, reducing the cost of data transfer to/from cloud providers.

- Reduced required network bandwidth as clients cache data locally, reducing the cost of data transfer to/from cloud providers.

◇ Handles high latency scenarious like mobile broadband, unstable networks, or when the end-user is far away from the cloud region where a Software-as-a-Service (SaaS) or app is deployed.

◇ Uniquely enables SaaS companies to provide excellent service worldwide with as little as one cloud region.

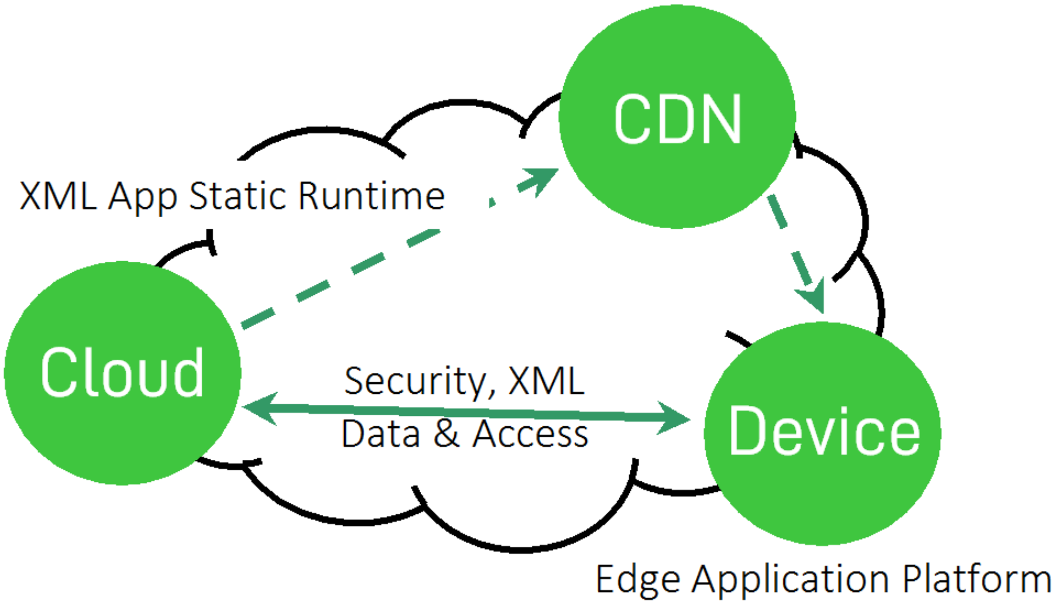
◇ The drawback is that developers need to develop in XML instead of JavaScript or server code, which for people without any prior XML knowledge might be challenging.

◇ Architectural patterns may need to change somewhat as the client-side edge execution favors backends designed as stateless micro services, i.e. a Service Oriented Architecture (SOA), instead of large monolithic APIs.

It is our experience that it is considerably faster to develop applications using XML, than creating the same application using traditional JavaScript frameworks and technologies. It results in considerably less source code that is easier to read and maintain. The source code can be validated by an XML schema, and it becomes much easier to collaborate between several people using GIT to build large enterprise-grade projects than having multiple people editing the same JavaScript code. The key is here how XML can be modularised using well-known technologies like XLink and split up into multiple documents and how XML enables a higher abstraction level with larger building blocks than traditional programming.

XML software applications can be hosted on any web server. They do not require any specific installed code in the cloud to work, nor are they dependent on any build toolchain. They can also be cached out close to end-users using any Content Delivery Network (CDN) as they are not dependent on server-side logic.

Figure 9. Uniquely leveraging a CDN to distribute XML software applications for low-latency



We thereby believe that applications on the Internet, that remind you of traditional software, will be more and more written using XML as they are easy to build, simple to host, and share a lot of properties of what made HTML popular in the first place. Almost anyone can write their own home page using HTML, but not everyone can build an advanced HTML5 application using JavaScript. XML software applications have the ability to bring the simplicity of markup back, replacing the programmatic web we see today with declarative markup.

Hopefully these two projects can show what is possible to do with XML, in order for more people to follow and build up a strong toolchain for web-based XML development. Together the available efforts and tools can be extended and packaged to make XML mainstream for web developers.

Imagine surfing a web-based of XML software applications and XForms instead of HTML and JavaScript.

Markup UK

Markup UK Proceedings as CSS

Tony Graham, Antenna House

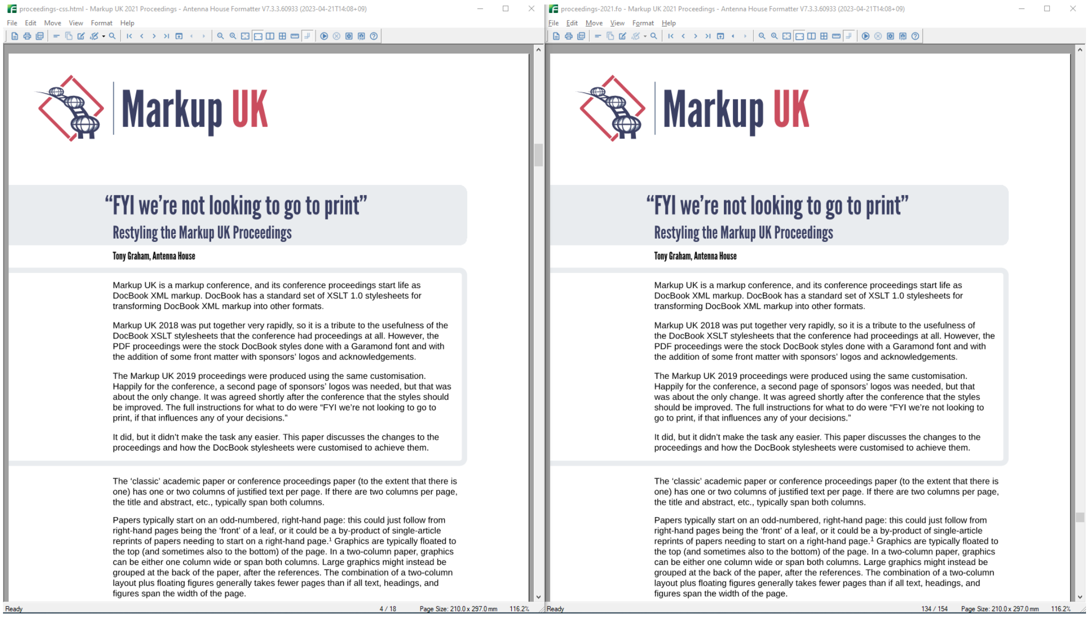
Markup UK conference proceedings start life as DocBook XML markup and are formatted as PDF using XSL-FO that is generated by the DocBook XSLT 1.0 stylesheets.

This paper discusses formatting the conference proceedings using CSS styles from the DocBook “xslTNG” XSLT 3.0 stylesheets.

As explained in *We’re Not Looking to go to Print* [GRAHAM2021], the 2018 and 2019 proceedings used almost stock XSLT 1.0 stylesheets to generate XSL-FO that was formatted using Antenna House Formatter, and the styles were then revised for Markup UK 2021 (and later) by using a more substantive customisation of the XSLT 1.0 stylesheets. Recreating the proceedings using CSS styles was an experiment in using CSS to format DocBook and, as it turned out, the first real test for using the DocBook xslTNG stylesheets to generate PDF.

As expected, it is possible to make the CSS version virtually identical to the existing XSL-FO version of a paper.[1]

Figure 1. Paper formatted using CSS (left) and XSL-FO (right)



proceedings-css.html - Markup UK 2021 Proceedings - Antenna House Formatter V7.3.3.60933 (2023-04-21T14:08+09)

Markup UK

“FYI we’re not looking to go to print”

Restyling the Markup UK Proceedings

Tony Graham, Antenna House

Markup UK is a markup conference, and its conference proceedings start life as DocBook XML markup. DocBook has a standard set of XSLT 1.0 stylesheets for transforming DocBook XML markup into other formats.

Markup UK 2018 was put together very rapidly, so it is a tribute to the usefulness of the DocBook XSLT stylesheets that the conference had proceedings at all. However, the PDF proceedings were the stock DocBook styles done with a Garamond font and with the addition of some front matter with sponsors’ logos and acknowledgements.

The Markup UK 2019 proceedings were produced using the same customisation. Happily for the conference, a second page of sponsors’ logos was needed, but that was about the only change. It was agreed shortly after the conference that the styles should be improved. The full instructions for what to do were “FYI we’re not looking to go to print, if that influences any of your decisions.”

It did, but it didn’t make the task any easier. This paper discusses the changes to the proceedings and how the DocBook stylesheets were customised to achieve them.

The ‘classic’ academic paper or conference proceedings paper (to the extent that there is one) has one or two columns of justified text per page. If there are two columns per page, the title and abstract, etc., typically span both columns.

Papers typically start on an odd-numbered, right-hand page: this could just follow from right-hand pages being the ‘front’ of a leaf, or it could be a by-product of single-article reprints of papers needing to start on a right-hand page.[1] Graphics are typically floated to the top (and sometimes also to the bottom) of the page. In a two-column paper, graphics can be either one column wide or span both columns. Large graphics might instead be grouped at the back of the paper, after the references. The combination of a two-column layout plus floating figures generally takes fewer pages than if all text, headings, and figures span the width of the page.

Ready 4 / 18 Page Size: 210.0 x 297.0 mm 116.2%

proceedings-2021.fo - Markup UK 2021 Proceedings - Antenna House Formatter V7.3.3.60933 (2023-04-21T14:08+09)

Markup UK

“FYI we’re not looking to go to print”

Restyling the Markup UK Proceedings

Tony Graham, Antenna House

Markup UK is a markup conference, and its conference proceedings start life as DocBook XML markup. DocBook has a standard set of XSLT 1.0 stylesheets for transforming DocBook XML markup into other formats.

Markup UK 2018 was put together very rapidly, so it is a tribute to the usefulness of the DocBook XSLT stylesheets that the conference had proceedings at all. However, the PDF proceedings were the stock DocBook styles done with a Garamond font and with the addition of some front matter with sponsors’ logos and acknowledgements.

The Markup UK 2019 proceedings were produced using the same customisation. Happily for the conference, a second page of sponsors’ logos was needed, but that was about the only change. It was agreed shortly after the conference that the styles should be improved. The full instructions for what to do were “FYI we’re not looking to go to print, if that influences any of your decisions.”

It did, but it didn’t make the task any easier. This paper discusses the changes to the proceedings and how the DocBook stylesheets were customised to achieve them.

The ‘classic’ academic paper or conference proceedings paper (to the extent that there is one) has one or two columns of justified text per page. If there are two columns per page, the title and abstract, etc., typically span both columns.

Papers typically start on an odd-numbered, right-hand page: this could just follow from right-hand pages being the ‘front’ of a leaf, or it could be a by-product of single-article reprints of papers needing to start on a right-hand page.[1] Graphics are typically floated to the top (and sometimes also to the bottom) of the page. In a two-column paper, graphics can be either one column wide or span both columns. Large graphics might instead be grouped at the back of the paper, after the references. The combination of a two-column layout plus floating figures generally takes fewer pages than if all text, headings, and figures span the width of the page.

Ready 134 / 154 Page Size: 210.0 x 297.0 mm 116.2%

[1] For an in-depth analysis of the similarities and differences between CSS and XSL-FO, see the *XSL-FO/CSS Comparison* available from `https://www.antennahouse.com/xsl-fo-css-comparison`.

1. DocBook

DocBook is “a schema (available in several languages including RELAX NG, SGML and XML DTDs, and W3C XML Schema) maintained by the DocBook Technical Committee of OASIS. It is particularly well suited to books and papers about computer hardware and software (though it is by no means limited to these applications).” [DOCBOOK] It has been under development since 1991, when it began as an SGML DTD for document interchange between HaL Computer Systems and O’Reilly & Associates. It was further developed and expanded by the Davenport Group before being taken on by the DocBook Technical Committee of OASIS in 1998.

As it is the best-known standard schema for technical documentation, DocBook or DocBook customisations are used for the proceedings of the Markup UK, XML Prague, and Balisage conferences.

2. DocBook Stylesheets

DocBook documents, as they are (usually) technical documents, are meant for human consumption. Companies developed their own tools for formatting DocBook[2], then as successive stylesheet standards became available, open source stylesheets for formatting or transforming DocBook were made available by Norman Tovey-Walsh. The original DSSSL stylesheets [DOCBOOKDSSSL] are no longer maintained; the XSLT 1.0 stylesheets are widely used and are actively maintained[3] by Bob Stayton [DOCBOOK-XSL] and others; the XSLT 2.0 stylesheets were never as ubiquitous as the XSLT 1.0 stylesheets, and their development was discontinued in July 2020 in favour of the XSLT 3.0 stylesheets. The “xslTNG” XSLT 3.0 stylesheets do not yet have the same user base or the same number of contributors as the XSLT 1.0 stylesheets, but the XSLT 1.0 stylesheets had a 16-year head start.

3. Comparing XSLT 1.0 and xslTNG Stylesheets

3.1. Localisation

The XSLT 1.0 stylesheets have localisations for generated text in over 70 languages. The xslTNG stylesheets reuse the localisation data to support the same languages.

3.2. Build system

The XSLT 1.0 stylesheets use Apache Ant to run all stages of generating output from DocBook, whereas the xslTNG stylesheets (following from the XSLT 2.0 stylesheets) uses Gradle to build and package the XSLT and other files that you use, and the xslTNG release that you build or download is run using either Java or Python. Recent releases also include an experimental Docker file so that you can make a Docker image for running the xslTNG stylesheets.

3.3. Syntax highlighting

The XSLT 1.0 stylesheets use XSLTHL [XSLTHL], which is an extension function implemented for multiple XSLT processors and which uses XML data files. However, as noted in [GRAHAM2021], XSLTHL development has stalled and new languages are rarely added. The xslTNG stylesheets use Pygments, which is an external Python program but which covers a large number of languages and which is still actively developed.

[2] I was formatting DocBook SGML into PostScript in the early 1990s.

[3] At the time of this writing, the GitHub repository lists 6,942 commits.

3.4. XSLT debugging

XSLT debugging is not always straightforward with either the XSLT 1.0 stylesheets or the xslTNG stylesheets. The XSLT 1.0 stylesheets can preprocess a document to either strip or add namespaces, depending on the document and the stylesheet being run, which makes it meaningless to try to get an XSLT debugger to break when a source node is processed because every node is preprocessed. The xslTNG stylesheets make multiple passes over the document to, for example, convert a DocBook 4 document into DocBook 5 and normalise titles to be contained in `<info>` elements. The xslTNG stylesheets make extensive use of `xsl:evaluate`, so it is not always possible to set a break point on the source node to be processed by a particular template.

With both the XSLT 1.0 stylesheets and the xslTNG stylesheets, the first advice is to add `xsl:message` messages or to generate comments or literal text that will show up in the result to indicate that a particular template was used. The xslTNG stylesheets support an `org.docbook.xsltng.verbose` Java system property to control debugging messages output by the extension functions.

3.5. Cover pages

The XSLT 1.0 stylesheets have an extensive mechanism for generating front and back covers, inside cover pages, etc. The xslTNG 2.1.4 stylesheets can be customised to generate cover pages and other front matter as part of the titlepage processing for a `book`, etc. The xslTNG 2.1.4 stylesheets support an `m:back-cover` mode that can be used to generate a back cover. Support for generating front and back covers is likely to be further developed in future releases.

3.6. PDF bookmarks

Both the XSLT 1.0 stylesheets and the xslTNG stylesheets can generate PDFs containing bookmarks.

4. Current status

The xslTNG stylesheets are a moving target. When this work started, the current release was 2.1.2. At the time of this writing, it is now 2.1.4, and by the time that you read this, the stylesheets will be further updated.

The stylesheets for generating paged media output were less well exercised than the stylesheets for producing web pages. The paged media stylesheets have been improved in response to issues raised while developing the CSS customisation. There are still some open issues, and it is likely that paged media support will be further improved shortly.

There was not much time available for developing the CSS customisation, and it is still a work in progress. The code is available in the `css-proc` branch on GitHub [MUK-xsl] for anyone to copy and modify. Comments and pull requests are welcome.

5. Acknowledgements

Norman Tovey-Walsh is developing the xslTNG stylesheets that are the subject of this paper. He made numerous improvements in response to issues raised, often with very quick turnaround.

Bibliography

[DOCBOOK] docbook.org: What is DocBook?. https://docbook.org/whatis

[DOCBOOKDSSSL] Norman Walsh: README for the DocBook Stylesheets. https://github.com/docbook/dsssl/blob/master/README.adoc

[DOCBOOK-XSL] Bob Stayton: DocBook XSL: The Complete Guide. Fourth Edition, September 2007, OASIS. http://www.sagehill.net/docbookxsl/index.html

[GRAHAM2021] “We’re Not Looking to go to Print”. Antenna House. https://markupuk.org/2021/webhelp/index.html#ar10.html

[MUK-docbook] Markup UK: ‘MUK_docbook’ add-on Oxygen framework, Markup UK https://github.com/MarkupUK/paperFramework

[MUK-xsl] Markup UK: XSL stylesheets for Markup UK proceedings, Markup UK https://github.com/MarkupUK/MUK-xsl

[XSLT10-STYLESHEETS] DocBook: DocBook XSLT 1.0 Stylesheets, DocBook https://github.com/docbook/xslt10-stylesheets

[XSLTHL] Michal Molhanec, Jirka Kosek, Michiel Hendriks: XSLT syntax highlighting, https://github.com/xmlark/xslthl

Markup UK

Enhancing Markup Quality Assurance with Automated Schema Visualization

Sven Reinck, FLUXparticle

The paper describes an XSD visualizer plugin that automates schema visualization and emphasizes the accuracy and completeness of element structure. It provides a comprehensive view of the inheritance hierarchy and actual element structure, allowing users to quickly identify potential errors and ensure the correctness and completeness of markup documents. This tool is useful for academics, software engineers, and end-users interested in enhancing their markup quality assurance practices through automated schema visualization with a specific focus on element structure.

1. Introduction

Schema development and processing play a pivotal role in ensuring the quality and integrity of structured data. However, several challenges arise in this process, hindering the effective management and optimization of schemas. This paper aims to address these challenges by introducing the "XSD Visualizer Plugin", a comprehensive tool designed to enhance schema quality assurance.

First, it is important to understand that the use of XML can be divided into two different fields of use that make different requirements of XML schemas and pose distinct challenges for the quality. One is structured data and the other is document content. Both provide structured representation and interoperability, but are completely different in the features they demand from a schema.

On the one hand, structured data is often defined by a hierarchy of data types. As data formats evolve and requirements change, modifying elements within a hierarchy can be a challenge. A minor misstep can have a cascading effect on the entire data structure.

On the other hand, document content often allow the same types of elements on different levels of the structure but not in the same way, it can be very difficult to describe such a schema in the same way as a data format. Which often leads to very complex schemas that are not really helpful to understand the format. That's why e.g. Schematron takes a completely different approach to describe XML.

So this paper will mostly focus on structured data and will only present rough ideas on how to visualize the structure of document content. One of the primary challenges faced by developers and users of structured data is comprehending the schema as a whole. Schemas often comprise complex hierarchies of types, making it difficult to gain a clear and holistic understanding of their complete structure. This lack of comprehensive understanding can impede effective schema modification, validation, and overall maintenance.

To address these challenges, the "XSD Visualizer Plugin" offers a solution that is more intuitive and seamless than traditional schema visualization tools. Because it is a plugin it integrates directly into the user's usual working environment. And since its focus is visualization, the user experience is free from distractions.

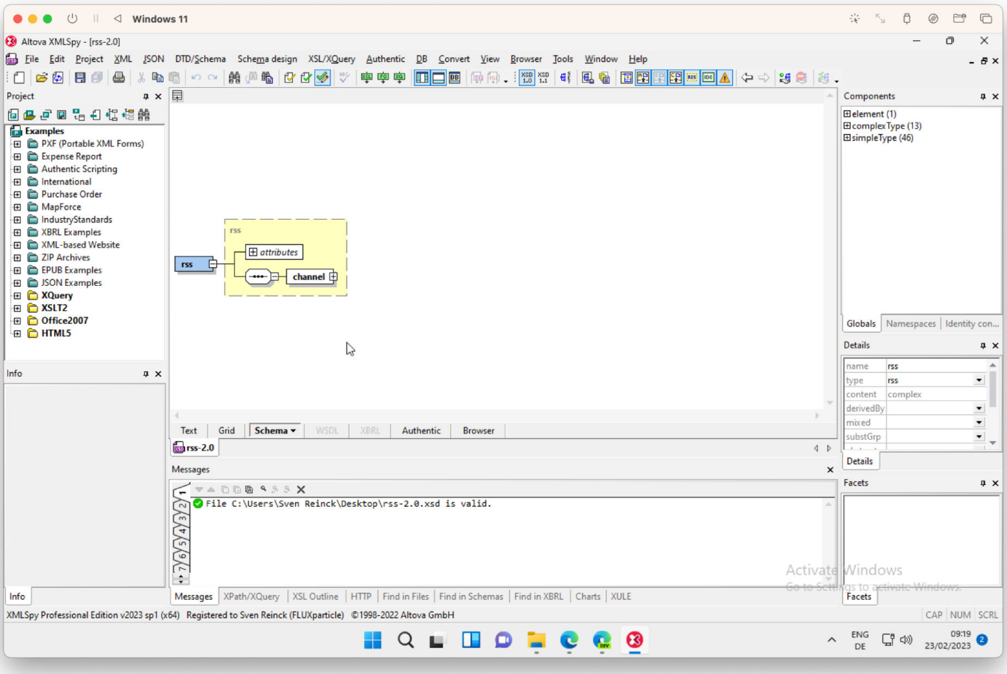
Additionally, the "XSD Visualizer Plugin" shows the effective structure of elements while preserving the information how it was created, allowing users to modify elements within the hierarchy and maintaining schema correctness. This feature makes it easier to change the schema without creating collateral damage and compromising the overall quality of the schema.

In the subsequent sections of this paper, we will delve deeper into the key features and functionalities of the "XSD Visualizer Plugin", illustrating how it addresses these challenges and improves schema quality assurance.

2. Background and Related Work

In the realm of XML schema visualization and editing, several existing tools have aimed to provide support for navigating and understanding complex schemas. Oxygen XML Editor, Altova XmlSpy, and Liquid XML Studio are among the notable tools in this domain. While these tools have made significant contributions, over time they have gained so much complexity that they can be quite intimidating on first use and due to the density of features the main purpose of a visualizer software - giving a clear overview - gets lost.

Here is a screenshot of XMLSpy, for example. When opening the RSS format XML schema. While the visualization of the schema is a good start, there is a lot going on around it that can be quite intimidating and lets the user question where to go next.

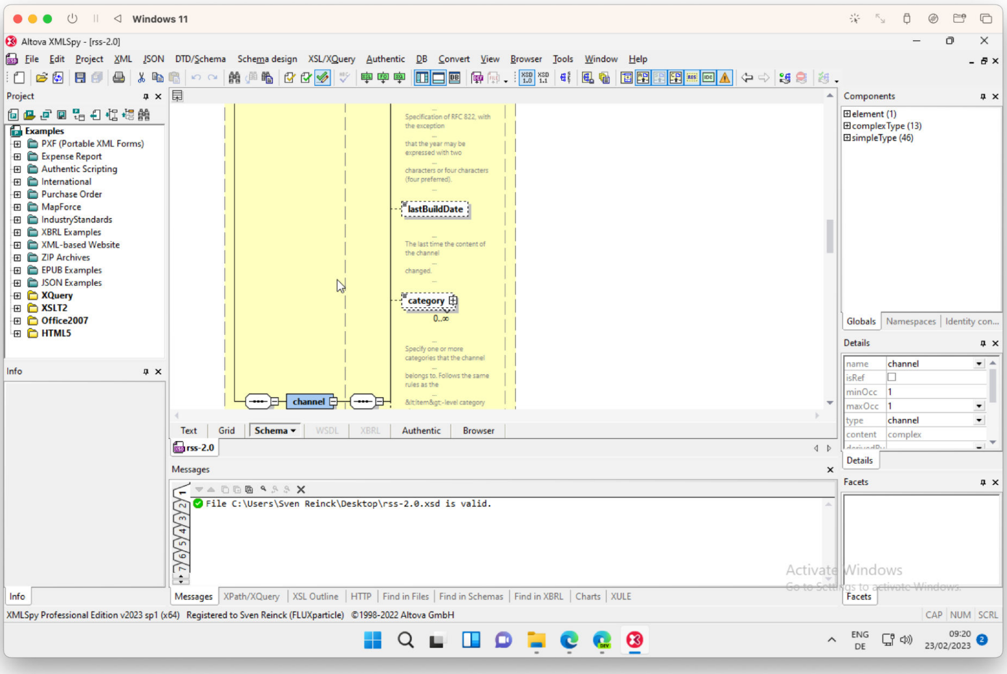


name	rss
type	rss
content	complex
derivedBy	
mixed	
substGrp	

```
File C:\Users\Sven Reinck\Desktop\rss-2.0.xsd is valid.
```

XMLSpy Professional Edition v2023 sp1 (x64) Registered to Sven Reinck (FLUXparticle) ©1998-2022 Altova GmbH

In the next screenshot the channel element is open but because of the annotations the graphic gets so big that it cannot be shown entirely on the screen. While there is an option to hide the annotations, this is the default behavior and the user has to search for the option to make the visualization more readable.



File C:\Users\Sven Reinck\Desktop\rss-2.0.xsd is valid.

name	channel
isRef	□
minOcc	1
maxOcc	1
type	channel
content	complex

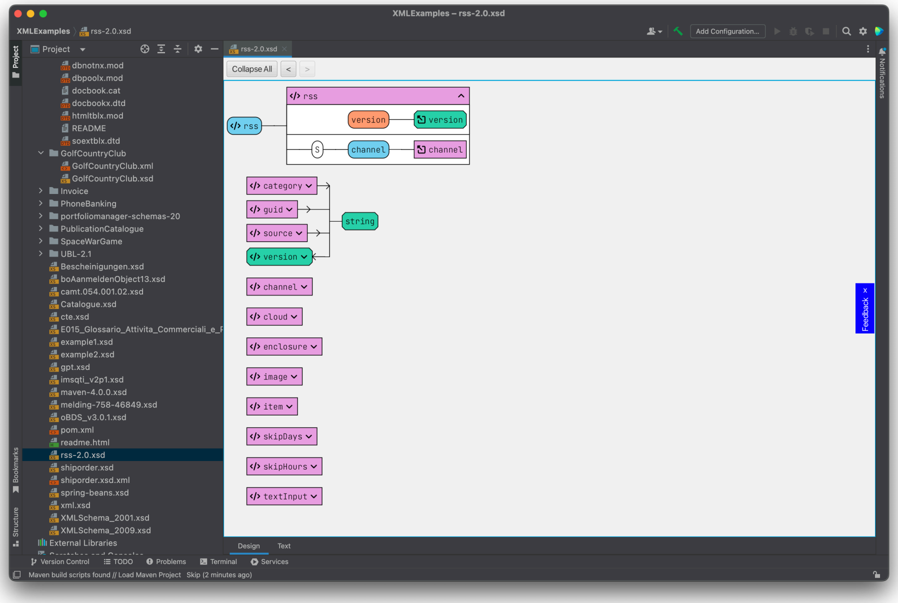
XMLSpy Professional Edition v2023 sp1 (x64) Registered to Sven Reinck (FLUXparticle) ©1998-2022 Altova GmbH

3. Introducing the XSD Visualizer Plugin

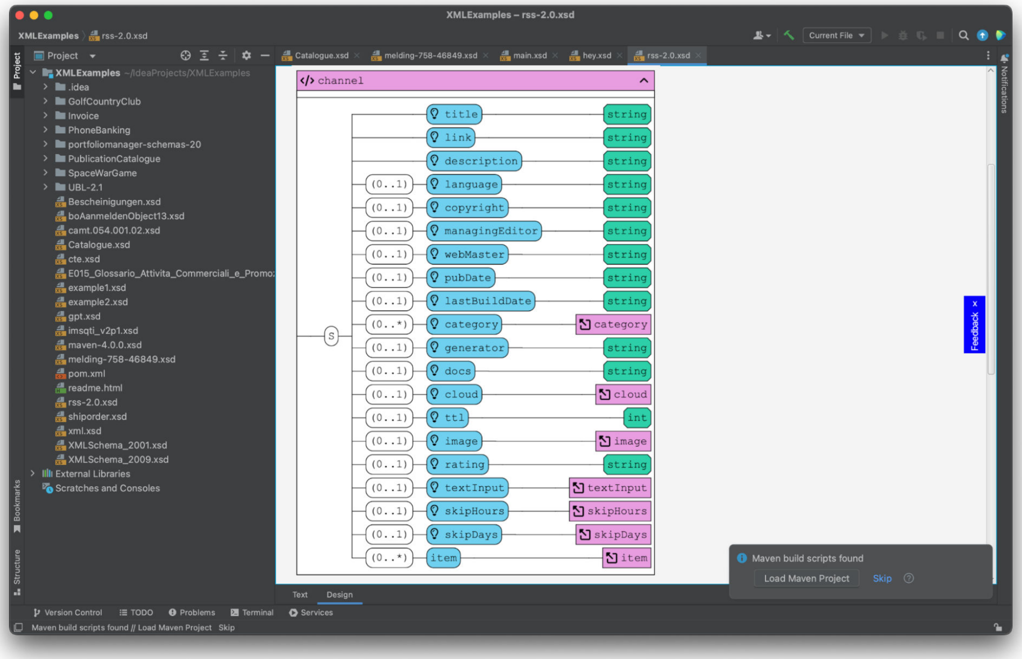
By focusing on visualization - for the cost of editing features at this moment - the "XSD Visualizer Plugin" aims to provide a solution that mitigates the steep learning curve often associated with these traditional schema visualization tools. Recognizing the importance of a user-friendly and intuitive experience, the "XSD Visualizer Plugin" prioritizes simplicity and ease of use.

Here are the same screenshots as before with the "XSD Visualizer Plugin":

The interface is much cleaner. Mostly because it's the UI of IntelliJ. But the plugin focuses mainly on the visualization.



The channel type has an arrow in front of it. When clicked, it doesn't open like in XMLSpy but instead moves the further down the already visible channel type and opens it. The complete type fits on the screen because the annotations are only shown as a tooltip when the user hovers the mouse cursor over the name. The presence of an annotation is shown by the light bulb.



By streamlining the user interface and intuitive interaction design, the "XSD Visualizer Plugin" enables users, including beginners, to quickly grasp the functionalities and harness the power of schema visualization. The visual representation of the schema's inheritance hierarchy, coupled with a seamless navigation experience, empowers users to understand the schema as a cohesive whole without struggling through complex learning curves. While its current scope is limited to IntelliJ and XML Schema, future developments will be explored and discussed in a subsequent section.

4. Key Features for Assessing Schema Quality

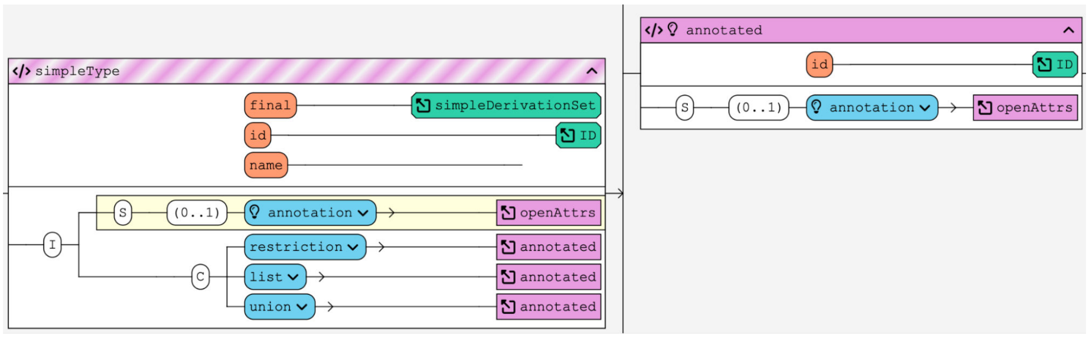
The "XSD Visualizer Plugin" offers a range of features that contribute to improving the quality and understanding of XML schemas. These features provide valuable insights into the inheritance structure, effective element and type structure, and enable editing through the Jump-to-Code functionality. Other features are still in development and will be discussed in detail in a dedicated section.

4.1. Visualizing Inheritance Structure

One of the key features of the "XSD Visualizer Plugin" is its ability to visually represent the inheritance structure of XML schemas. By analyzing the schema's hierarchy and considering all parent types, the plugin generates a clear and comprehensive visualization that highlights the relationships and dependencies between elements and types. This visualization allows users to understand the inheritance chain, facilitating the identification of potential issues or conflicts within the schema's structure.

4.2. Providing Effective Element and Type Structure

The plugin not only displays the inheritance structure but also provides an effective view of the element and type structure within the schema. It presents the actual structure of elements, taking into account the contributions from all parent types. This ensures that users have a complete understanding of how elements are defined and organized, enabling them to make informed decisions when working with complex schemas.



4.3. Jump-to-Code for Editing

The "XSD Visualizer Plugin" introduces a Jump-to-Code feature, which allows users to navigate directly to the code corresponding to a specific element or type within the schema. This functionality enables easy modification, even while the full editing features are still in development. Users can conveniently locate the code associated with a particular element, make necessary changes, and observe the immediate impact on the visual representation of the schema.

4.4. Future Development

While the aforementioned features contribute significantly to improving schema quality, it is important to note that the "XSD Visualizer Plugin" is an evolving tool. The development team is actively working on additional features and enhancements to further enhance schema quality assurance. These forthcoming features will be explored in depth in a subsequent section, providing insights into the future roadmap and the potential benefits they bring to schema designers and developers.

5. Enhancing the Schema Quality Workflow with Work-in-Progress Features

The "XSD Visualizer Plugin" is continuously evolving to provide a comprehensive suite of features that enhance the workflow for ensuring schema quality. In addition to the existing capabilities, the plugin is currently under development with two exciting features aimed at improving the understanding and composition of XML-based document formats: the TreeView and the Composite View.

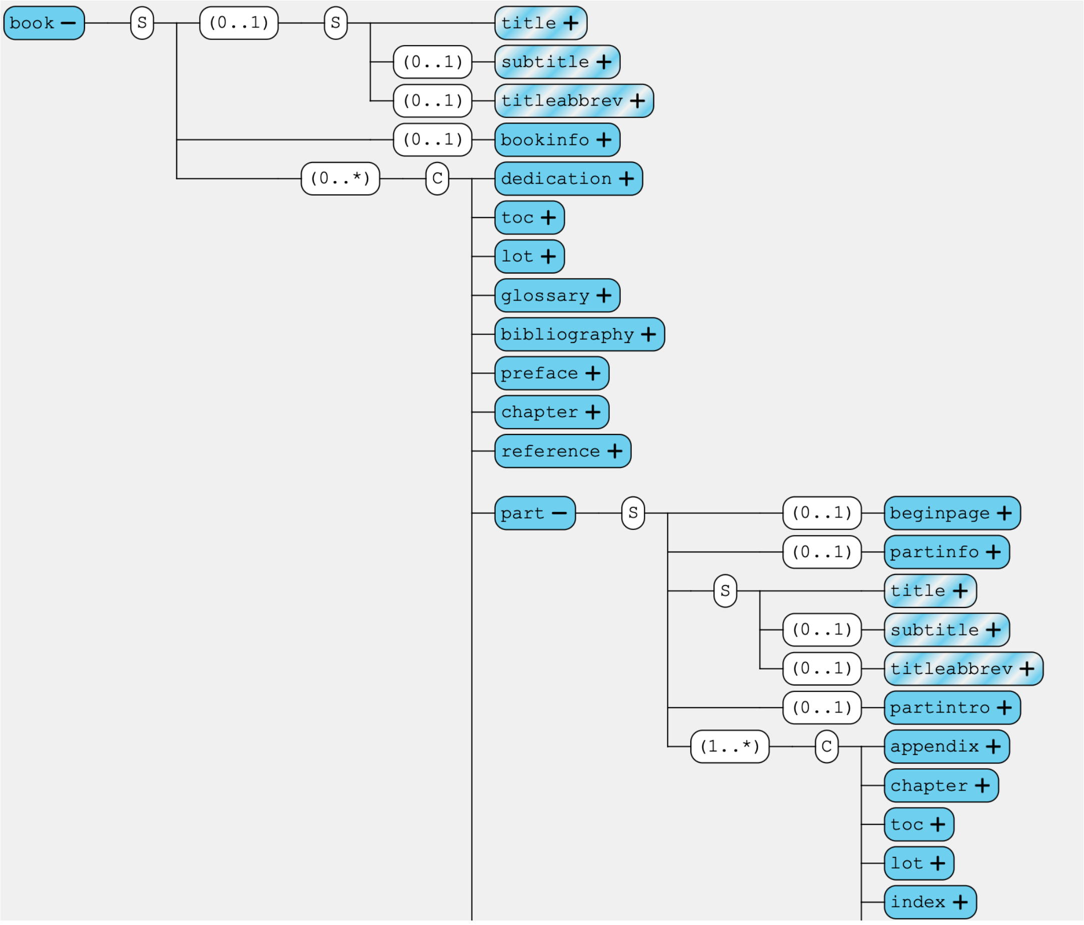
5.1. Tree View for Exploring the Actual Structure

The Tree View feature is being developed to enable users to explore the actual structure of an XML file in alignment with an associated XML Schema Definition (XSD) or Document Type Definition (DTD). This feature provides a visual representation of the hierarchical structure of the XML format, showcasing the relationships between elements, attributes, and their respective values. By visualizing the possible structure of the XML file alongside the XML file, users can easily identify any discrepancies or inconsistencies between the file and its associated schema, ensuring adherence to the defined structure.

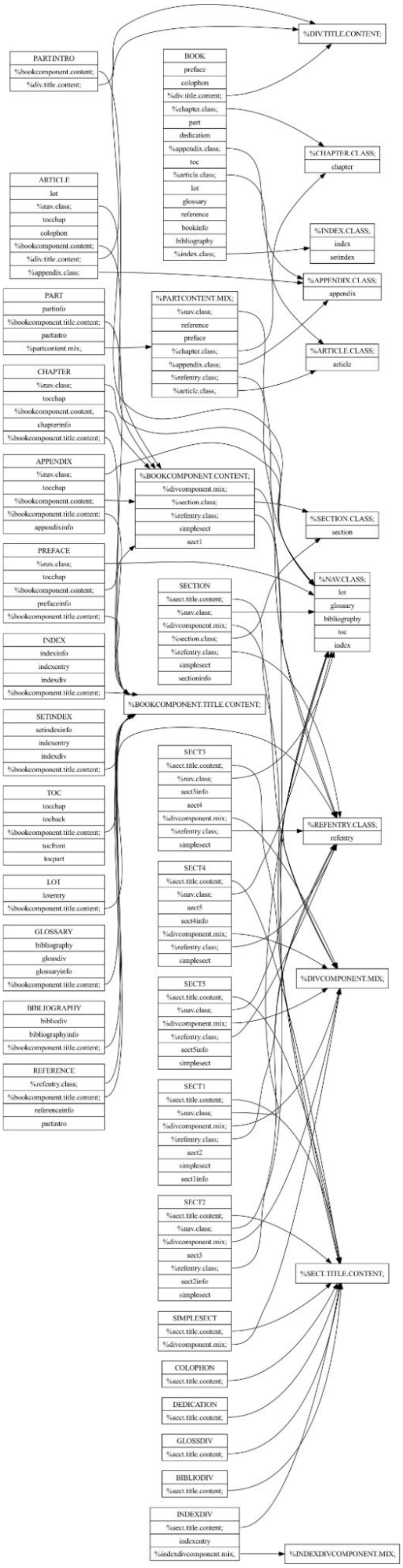
This is a screenshot of the DocBook structure. It is easy to see which elements are optional "0..1" and which are choices "C". The "S" means sequence, so in this case "title", "subtitle" and "titleabbrev" have to be at the top of the "book" element and in this order. The half filled boxes mean that these elements allow text directly in them.

A click on the plus sign reveals what child elements can appear inside this element.

Tree View for Exploring the Actual Structure



5.2. Composite View for Understanding Complex Composition Structures



The Composite View is another work-in-progress feature that aims to facilitate a deeper understanding of the complex composition structures present in XML-based document formats. Many XML document formats, such as DocBook, involve intricate relationships and dependencies between various components. The Composite View will provide a visual representation of these composition structures, showcasing the interconnections, dependencies, and hierarchy of components within the XML-based document format. This feature will aid users in comprehending the complex relationships between different parts of the document, thereby improving their ability to design, validate, and maintain high-quality schemas for such formats.

Since DocBook 4.5 is an XML-based document format that is described as DTD - a format that is better for data formats - the structure can be quite complicated. This is an idea on how to visualize the composition of these groups so the authors of this format have a better understanding of their format. I assume it can be pretty hard to keep such a structure in your head. Maybe this visualization can even help to clean up the format and make it easier.

These work-in-progress features demonstrate the ongoing commitment of the "XSD Visualizer Plugin" team to enhance the schema quality workflow. By introducing the Tree View and Composite View, the plugin aims to provide users with powerful tools to explore and understand the actual structure of XML files, ensuring alignment with the associated schema and enabling efficient handling of complex composition structures.

6. Implementation Details of the XSD Visualizer Plugin

The "XSD Visualizer Plugin" is implemented using a combination of Java and Kotlin programming languages, leveraging the JavaFX framework for its user interface (UI) components. However, due to the underlying Swing-based UI framework used by IntelliJ, the plugin incorporates a compatibility layer to ensure seamless integration within the IntelliJ environment. Nevertheless, the plugin is designed in such a way that future porting to Swing can be achieved with minimal effort, allowing for the removal of the compatibility layer.

The foundation of the "XSD Visualizer Plugin" is built upon Javis (Java Visualizer), a tool developed by the author for visualizing the execution of Java programs during training sessions. By leveraging the capabilities and insights gained from Javis, the plugin offers a robust and feature-rich visualization experience for XML schemas within the IntelliJ IDE.

To enable fluid and dynamic changes to the UI during the exploration and editing of schemas, the XSD Visualizer Plugin utilizes the Fenja library, developed by the author. Fenja provides a set of powerful tools and utilities specifically designed to facilitate a seamless and interactive user experience when working with dynamic UI elements. This library allows users to navigate, explore, and modify the schema in a fluid manner, ensuring a smooth and efficient workflow.

7. Feedback and Future Development

The "XSD Visualizer Plugin" has received valuable feedback from users and the development team remains committed to further improving and expanding its capabilities. If you also want to contribute to the roadmap of this plugin, please fill out this survey [https://xsdvisualizer.fluxparticle.com/permalink/ABe2S6LAAAA]. In response to user needs and emerging industry trends, the following future developments and enhancements are planned:

7.1. Support for Additional Schema Formats

Recognizing the diverse landscape of schema languages, the "XSD Visualizer Plugin" aims to broaden its support beyond XML Schema. The development team is actively working on incorporating support for other popular schema formats such as Schematron and Relax NG. Even JSON Schema is on the roadmap. This expansion will enable users to visualize and analyze schemas written in these languages, enhancing the overall versatility and usefulness of the plugin.

7.2. Potential Name Change to "SchemaViz"

As the "XSD Visualizer Plugin" evolves to support various schema formats, the plugin's name will probably transition to "SchemaViz." This name change would reflect its expanded scope and emphasize its applicability to a wider range of schema languages. The new name will align with the plugin's goal of providing comprehensive schema visualization capabilities, regardless of the underlying schema format.

7.3. Port to Visual Studio Code

In recognition of the popularity and widespread adoption of Visual Studio Code as a versatile and powerful code editor, the development team is actively pursuing a port of the "XSD Visualizer Plugin" to this platform. Leveraging the implementation in Kotlin, which can be compiled to JavaScript, the team is in the late stages of development for a Visual Studio Code version of the plugin. This will extend the reach of the plugin to a broader audience and provide seamless schema visualization within the Visual Studio Code environment.

7.4. HTML Export with SVG Graphics

To facilitate sharing and collaboration, the "XSD Visualizer Plugin" is introducing an HTML export feature. Users will be able to export parts of their schemas as HTML files, accompanied by SVG graphics that capture the visual representation of the schema. This export capability will enable users to generate visually appealing non-interactive representations of their schemas, which can be easily shared, viewed, and analyzed in web browsers or other compatible platforms.

8. Conclusion

In conclusion, the "XSD Visualizer Plugin" offers a powerful and intuitive solution for enhancing schema quality assurance and visualization within the IntelliJ IDE. By addressing the challenges associated with understanding and modifying XML-based document and data formats, the plugin provides users with a comprehensive set of features that promote efficient and effective schema design, validation, and maintenance.

Through its innovative visualizations, including the visualization of inheritance structures and effective element and type representations, the "XSD Visualizer Plugin" enables users to gain deep insights into their schemas. The clear and intuitive interface empowers users to identify and address quality issues, ensuring adherence to defined structures and promoting best practices in markup design.

The plugin's work-in-progress features, such as the Tree View for exploring the actual structure of XML files and the Composite View for understanding complex composition structures, further enhance the schema quality workflow. These features facilitate a holistic view of schemas, enabling users to navigate and analyze their structures with ease.

Looking ahead, the "XSD Visualizer Plugin" has ambitious plans for future development. The team is actively working on expanding support for other schema formats, such as Schematron, Relax NG and even JSON Schema, to cater to a broader range of user needs.

The plugin's portability to Visual Studio Code and the introduction of an HTML export feature with SVG graphics further emphasize its commitment to facilitating collaboration and sharing within the schema design community.

Overall, the XSD Visualizer Plugin provides a seamless and user-friendly experience for schema designers, developers, and stakeholders. It empowers users to navigate, explore, and validate their schemas with confidence, ultimately contributing to improved quality and efficiency in markup design.

Markup UK

Improving quality-critical XML workflows with XProc 3.0 pipelines

Achim Berndzen, <xml-project />

Thorsten Rohm, Thieme Compliance GmbH

Orchestrating complex XML pipelines has been a major topic of XML-related software development over the years. Comprehensive techniques have been developed to:

1. Deliver high-quality results
2. Ensure that the pipelines can be maintained
3. Allow the pipelines to be debugged for straightforward troubleshooting

The quality demands for the workflow and the produced results can vary: For example, you may find a very maintainable pipeline producing documents with very low quality demands, e.g. the system producing the static website for your local sports club. On the other hand you might find documents with very high quality demands produced by a pipeline that is not easy to maintain and debug. And, of course, the relationship between the quality of the documents and the maintainability of the pipeline producing these documents may change over time. Implementing new quality demands for the documents might have a negative impact on the pipelines quality. And sometimes in the history of developing a pipeline expected to produce documents with high quality demands, you might even decide to start over, as new quality demands for the documents threaten to impair the quality of your pipeline.

In this paper, we would like to report about a shared project of our two companies. We had to add new features to a well-established workflow producing documents in the medical sector that come with very high quality demands. As the existing workflow already had some pain points, we decided to start over and to refactor it. And we even decided to change the basic orchestrating technology: Since the existing workflow was based on a combination of Windows batch files calling different programs and some very elaborate XSLT stylesheets, we decided to use XProc 3.0 to orchestrate the workflow, thus doing away with as much shell scripting as possible while keeping the XSLT stylesheets to do the actual transformations.

As XProc 3.0 is a relatively new technology for orchestrating document workflows, we think our project might be of some interest to people developing and/or maintaining pipelines for documents with high quality demands. We will first provide some background context for the produced documents and their actual usage to elaborate the specific quality demands. This will be followed by an overview of the existing workflow and a discussion on its pain points and new demands. We will then give an overview of the new XProc 3.0 pipeline developed in the project and discuss some aspects of the used technology. The paper[1] concludes with the lessons learned in our project

[1] We would like to thank the reviewers of our abstract for their very helpful comments. A special thank goes out to Geert Bormans whose thoughtful remarks on the abstract helped to improved this paper significantly.

and the key takeaways of our project in a more general context of pipelines producing documents with high quality demands.

1. Introduction and background

1.1. About Thieme Compliance GmbH and patient education leaflets

Thieme Compliance GmbH, based in Erlangen, Germany, is a company that specialises in providing patient education solutions for healthcare facilities. These solutions include, among other things, information materials that educate patients about their illnesses, treatment options and possible risks. Patient education is an important aspect of healthcare as it helps patients make informed decisions about their health. It is also an important legal and ethical principle in medicine.

Thieme Compliance GmbH supports healthcare facilities in implementing this principle by providing customised information materials tailored to the specific needs of patients. The materials are developed in close cooperation with more than 400 experts from the medical community and tested for their comprehensibility and usefulness. Furthermore, a team of legal advisors ensures that the patient education content always corresponds to current case law. Professional societies and associations recommend the patient education leaflets from Thieme Compliance. In total, more than 2,000 patient education leaflets from more than 30 speciality areas are available in up to 31 languages. They are available in digital form as well as various print formats.

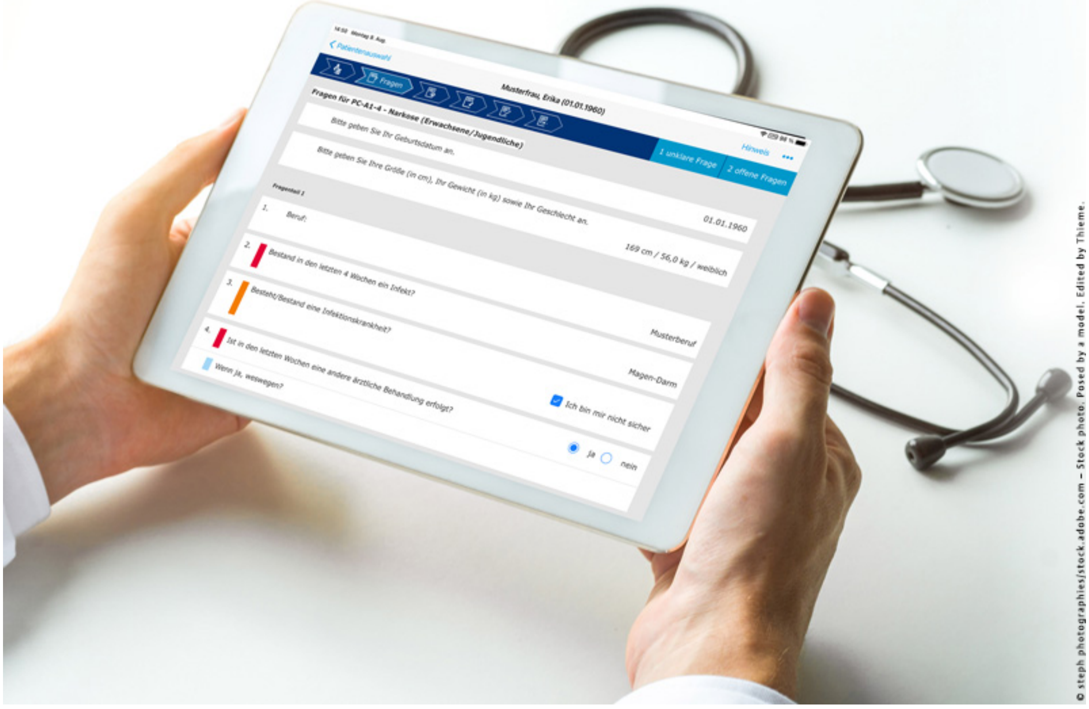
Figure 1. Patient education leaflet

© Thieme

The aim is to help ensure that patients are better informed and educated so that they can make decisions about their health in close cooperation with their doctors. Clinics and practices are supported in meeting legal requirements and minimising liability risks by using patient education leaflets. Under certain conditions, the clinics and practices can even receive a reduced insurance premium if they use the patient education leaflets from Thieme Compliance GmbH, as the risk of being sued by a patient is reduced.

The educational content is made available to clinics and practices via the self-developed software E-ConsentPro. E-ConsentPro is installed on-premise in clinics and practices and offers interfaces to, or is even embedded in, clinical information systems. It ships with Saxon-EE, Antenna House Formatter, XSLT as well as XSL-FO stylesheets, fonts, etc. and can thus be used to generate various media forms of the patient education leaflets, supplemented and personalised with data from the clinical information system.

Figure 2. "Anamnese mobil" app from E-ConsentPro



Patientenauswahl
Fragen
Musterfrau, Erika (01.01.1960)
Fragen für PC-A1-4 - Narkose (Erwachsene/Jugendliche)
Hinweis
1 unklare Frage
2 offene Fragen
Bitte geben Sie Ihr Geburtsdatum an.
01.01.1960
Bitte geben Sie Ihre Größe (in cm), Ihr Gewicht (in kg) sowie Ihr Geschlecht an.
169 cm / 56,0 kg / weiblich
Fragenteil 1
1. Beruf:
Musterberuf
2. Bestand in den letzten 4 Wochen ein Infekt?
Magen-Darm
3. Besteht/Bestand eine Infektionskrankheit?
☑ Ich bin mir nicht sicher
4. Ist in den letzten Wochen eine andere ärztliche Behandlung erfolgt?
● ja ○ nein
Wenn ja, weswegen?

© steph photographies/stock.adobe.com – Stock photo. Posed by a model. Edited by Thieme.

For parts of the patient education leaflets, such as the medical history, HL7's FHIR (Fast Healthcare Interoperability Resources) questionnaire resource is used for syntactic healthcare interoperability. The HL7 FHIR standard is based on a RESTful API architecture. This is an emerging standard for exchanging medical data between different systems and institutions and will replace the established HL7 V2 standard in the future. To ensure semantic healthcare interoperability as well, the questionnaire resource contains coding from SNOMED CT (Systematized Nomenclature of Medicine and Clinical Terms) or LOINC (Logical Observation Identifiers Names and Codes):

```
<fhir:item>
   <fhir:linkId value="MF_Erkrankungen_Familie__Erkrankung_Blutsverwandtschaft" />
   <fhir:text value="Among your blood relatives, are or were there any diseases or indications of a disease?" />
   <fhir:type value="open-choice" />
   <fhir:required value="true" />
   <fhir:repeats value="true" />
   <fhir:answerOption id="MF_Erkrankungen_Familie__Erkrankung_Blutsverwandtschaft__nein">
      <fhir:extension url="http://hl7.org/fhir/StructureDefinition/questionnaire-optionExclusive">
         <fhir:valueBoolean value="true" />
      </fhir:extension>
      <fhir:valueCoding>
         <fhir:system value="http://snomed.info/sct" />
         <fhir:version value="http://snomed.info/sct/900000000000207008/version/20220430" />
         <fhir:code value="160266009" />
         <fhir:display value="No family history of clinical finding (situation)" />
      </fhir:valueCoding>
   </fhir:answerOption>
   <fhir:answerOption id="MF_Erkrankungen_Familie__Erkrankung_Blutsverwandtschaft__Krebs">
      <fhir:valueCoding>
         <fhir:system value="http://snomed.info/sct" />
         <fhir:version value="http://snomed.info/sct/900000000000207008/version/20210131" />
         <fhir:code value="275937001" />
         <fhir:display value="Family history of cancer (situation)" />
      </fhir:valueCoding>
   </fhir:answerOption>
   <!-- ... -->
   <fhir:answerOption id="MF_Erkrankungen_Familie__Erkrankung_Blutsverwandtschaft__Erbkrankheiten">
      <fhir:valueCoding>
         <fhir:system value="http://snomed.info/sct" />
         <fhir:version value="http://snomed.info/sct/900000000000207008/version/20220430" />
         <fhir:code value="429962007" />
         <fhir:display value="Family history of hereditary disease (situation)" />
      </fhir:valueCoding>
   </fhir:answerOption>
</fhir:item>
```

The patient education content is created and managed using the component content management system Content Lifecycle System (CLS) from Empolis Solutions GmbH. It consists of XML files with single-source-capability, modularised using XInclude. Because of the rich semantics, a self-developed data structure T_0 XSD is used.

The patient education leaflets are published in various media formats such as PDF, XHTML, HTML5, WordML, SpreadsheetML, among others. To facilitate the data exchange, especially with partners, the content is converted into a variety of XML dialects as well as JSON formats. Therefore, fully or at least highly automated publishing pipelines are being developed and maintained.

1.2. About <xml-project /> and XProc

XProc 3.0 is a pipeline language with an XML syntax. It is based on XProc (1.0), which became a W3C recommendation in 2010. Based on user experience, a group of volunteers have worked together since 2017 as an W3C community group to improve and expand the original language. In September 2022, a community report on the core language specifications and the standard step library was published. While additional step libraries, e.g. for file processing, document validation, and paged media creation, are technically still under construction, we consider them to be very mature and in their final state. In fact, the project presented here relies heavily steps from the additional libraries, and they proved to be very useful and robust.

For those familiar with the original XProc, it might be interesting to mention some of the changes made for XProc 3.0. The most visible change is the expansion of the basic document model from XML only to a more realistic model for the latest processing: “Native” documents in XProc 3.0 are now XML, HTML, JSON, as well as text documents and binary documents (such as images or PDFs). The newly supported document types are accompanied by corresponding steps so that they can be used effectively in the pipelines. Further highlights of XProc 3.0 are the move to XPath 3.1 as the underlying processing language, XDM typing for options and variables along with a number of minor syntax tweaks that greatly improve the coding and debugging experience from the original XProc.

For those not familiar with XProc 1.0, or those who want to start over with XProc 3.0, there is now an improved learning base. Foremost, there is Erik Siegel's excellent book [Siegel:2020]. Erik also published a series of articles introducing XProc 3.0 on XML.com ([Siegel:2019], [Siegel:2020a], [Siegel:2020b]). For those who prefer videos, a series of six talks from Markup UK 2020 are available on the conference's YouTube channel. In addition to two talks on the basics of XProc 3.0, there are also talks on handling JSON documents, text documents and Zip archives.

Currently, two XProc 3.0 processors are known to be available: XML Calabash 3.0 is in its final phase as a successor to the well-known XML Calabash, both developed by Norman Tovey-Walsh. This paper is based on MorganaXProc-III, which is the successor to the now retired MorganaXProc. Also developed by <xml-project />, this is a Java (or JVM) based implementation that, in addition to the core specification and the standard step libraries, also implements the file step library and most of the validation library, while also supporting Extensible Validation Report Language (XVRL). It has been around as a public beta since February 2020, received a lot of useful bug reports from users and was released as version 1.0 in September 2022. Since then, it has received monthly updates with bug fixes and feature enhancements. MorganaXProc-IIIse is an open-source product released under GPL 3.0. Coming later this year is a second, commercial edition called MorganaXProc-IIIee (Extended Edition). It provides support for almost all optional features of XProc 3.0, with complete coverage of the proposed step libraries as well as processor-specific steps such as image processing.

2. Introduction to existing batches

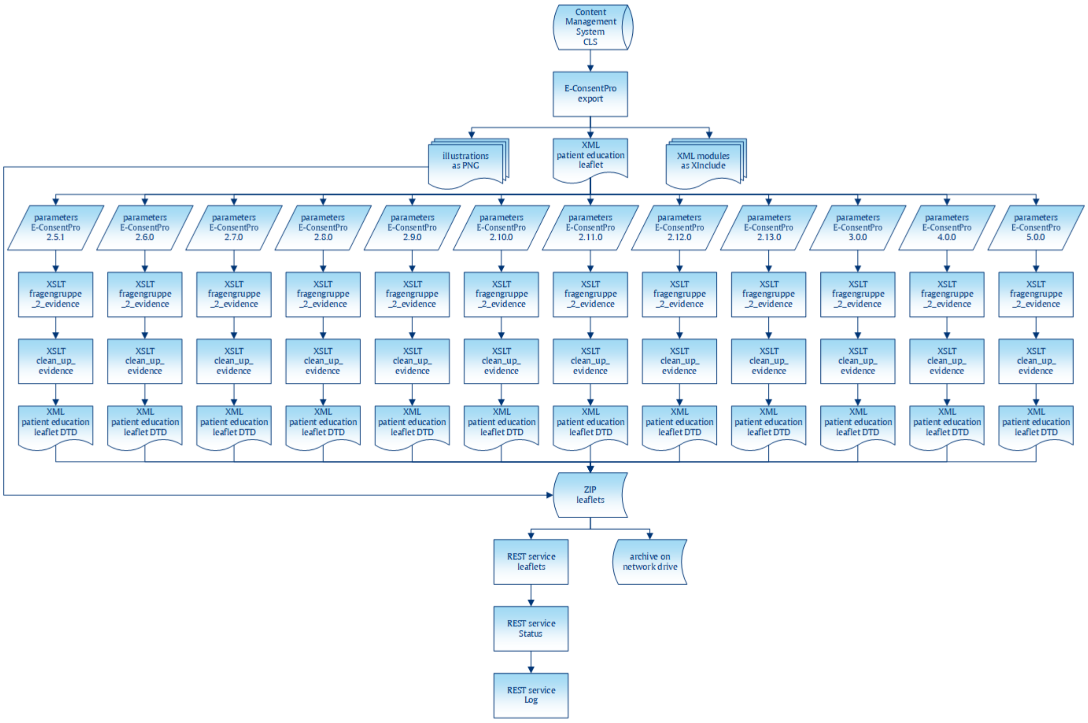
2.1. Batch “fragengruppe_2_evidence”

When it comes to the content delivery to Thieme Compliance's E-ConsentPro, the latest editions of the patient education leaflets are selected in the content management system CLS and an export is performed. The main XML file as well as all referenced XIncludes and images are exported from the database into a temporary folder on the CLS server. Here, the batch “fragengruppe_2_evidence” is used to

- ◇ Merge the main XML with the XIncludes
- ◇ Change the XML files from T_0 XSD to T_0 DTD by removing namespaces and inserting a Document Type Declaration
- ◇ Derive up to twelve different variants from the source XML file, each valid for the specific version of T_0 DTD employed for the different versions of E-ConsentPro currently in use in the market

When the transformation is complete, Beyond Compare from Scooter Software is used to copy the images from the source to the result folder. The output is then zipped together with the referenced images and delivered via REST services using curl.

Figure 3. Batch “fragengruppe_2_evidence”


Content Management System CLS
E-ConsentPro export
illustrations as PNG
XML patient education leaflet
XML modules as XInclude
parameters E-ConsentPro 2.5.1
parameters E-ConsentPro 2.6.0
parameters E-ConsentPro 2.7.0
parameters E-ConsentPro 2.8.0
parameters E-ConsentPro 2.9.0
parameters E-ConsentPro 2.10.0
parameters E-ConsentPro 2.11.0
parameters E-ConsentPro 2.12.0
parameters E-ConsentPro 2.13.0
parameters E-ConsentPro 3.0.0
parameters E-ConsentPro 4.0.0
parameters E-ConsentPro 5.0.0
XSLT fragengruppe _2_evidence
XSLT clean_up_ evidence
XML patient education leaflet DTD
ZIP leaflets
REST service leaflets
archive on network drive
REST service Status
REST service Log

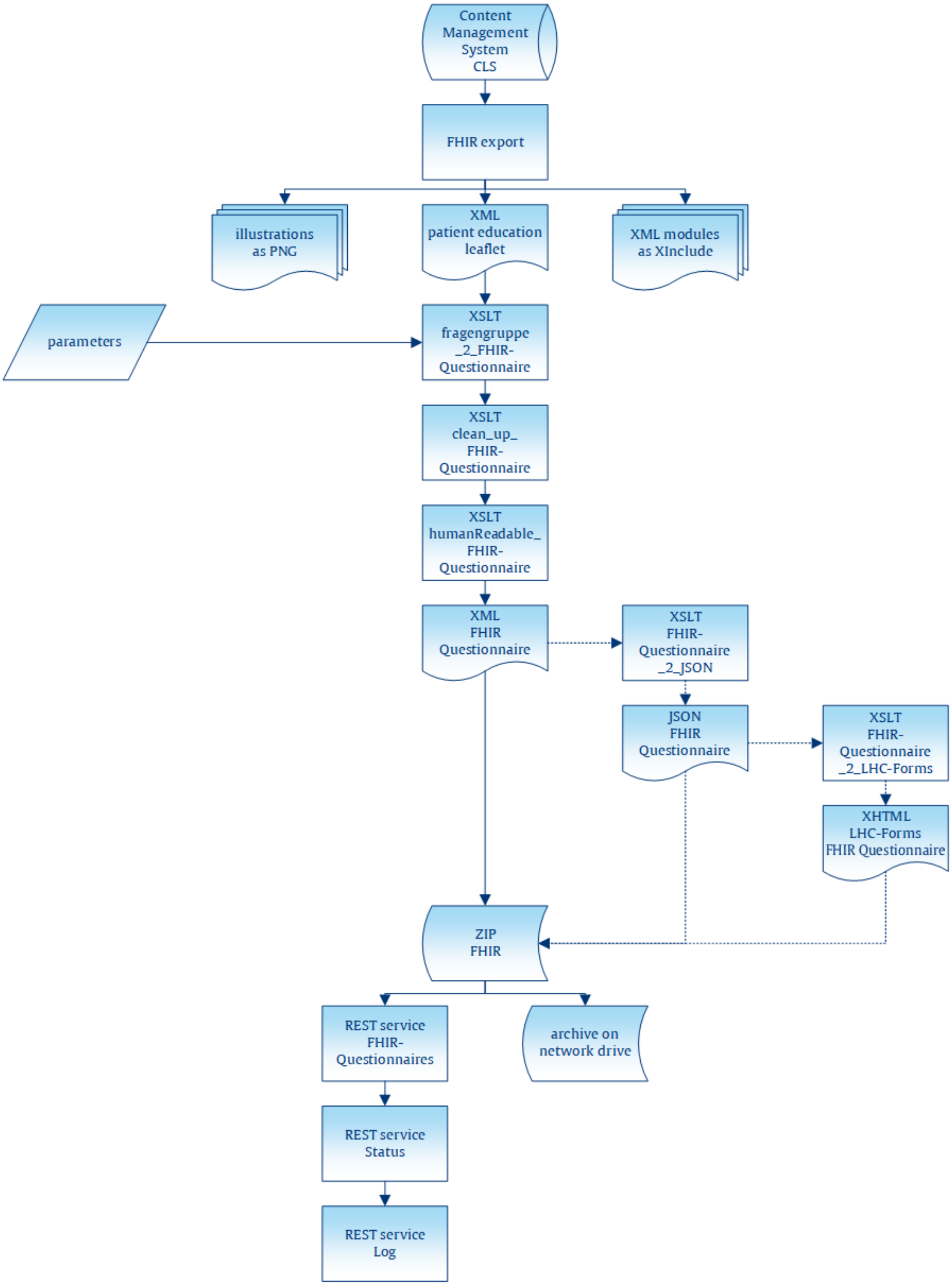
2.2. Batch “fragengruppe_2_FHIR-Questionnaire”

This second batch is used to deliver the medical history part of the patient education leaflet additionally as an HL7 FHIR questionnaire resource. Therefore, the main XML file as well as all referenced XIncludes are exported from the database into a temporary folder on the CLS server. A three-step transformation is then performed to:

- Merge the main XML with the XIncludes
- Generate the FHIR questionnaire resource and then clean it up (e.g. whitespace handling)
- Derive the human-readable part from the previously generated FHIR questionnaire resource and merge it back in
- Optionally transform the previously generated XML FHIR questionnaire resource into JSON
- Further optionally merge the just generated JSON FHIR questionnaire resource into an XHTML template for an output based on LHC-Forms

Afterwards, the generated results are zipped and delivered via REST services using curl. The images are not needed for this output and are therefore discarded.

Figure 4. Batch “fragengruppe_2_FHIR-Questionnaire”


Content Management System CLS
FHIR export
illustrations as PNG
XML patient education leaflet
XML modules as XInclude
parameters
XSLT fragengruppe _2_FHIR-Questionnaire
XSLT clean_up_ FHIR-Questionnaire
XSLT humanReadable_ FHIR-Questionnaire
XML FHIR Questionnaire
XSLT FHIR-Questionnaire _2_JSON
JSON FHIR Questionnaire
XSLT FHIR-Questionnaire _2_LHC-Forms
XHTML LHC-Forms FHIR Questionnaire
ZIP FHIR
REST service FHIR-Questionnaires
archive on network drive
REST service Status
REST service Log

3. Pain points of the existing batches

3.1. Lacking of flexibility for inserting additional XSLT steps (in between)

Regarding the batch “fragengruppe_2_evidence”, it all started about a dozen years ago with just one single XSLT stylesheet. It was called with an initial template and used `fn:collection()` to process the entire source folder.

Over the years, more and more requirements have been added as well as, and new E-ConsentPro versions were released that had to be supported with the matching version of the XML content. As a result, the original XSLT stylesheet became more and more complex. In addition, some of the new requirements could no longer be sensibly implemented within a single XSLT stylesheet – for example, additional whitespace handling following a transformation performed on the original stylesheet. The simplest way to extend the existing transformation, without having to adjust anything else, was by adding another XSLT stylesheet using `@saxon:next-in-chain`.

Using `@saxon:next-in-chain` was an incredibly easy and effective solution, but it also comes with some downsides. Each XSLT stylesheet is orchestrated from the previous one. Inserting an additional stylesheet is inflexible and requires an unrelated XSLT to be changed.

Saxonica has since deprecated `@saxon:next-in-chain` and suggests using `fn:transform()` instead. While we are convinced that this function is a suitable replacement for `@saxon:next-in-chain`, but we have not looked into this simply due to the additional pain points as well as the requested improvements for existing pipelines.

3.2. No easy way to debug the intermediate results of each XSLT step

Debugging a multi-step XSLT pipeline can be an arduous process. A no longer matching `<xsl:template>` in a later XSLT stylesheet, because an earlier stylesheet has already changed the `node()`, occurs frequently. Storing intermediate results from each step therefore is necessary but is not as easy as it could be, based on the existing batches. XSLT provides `<xsl:result-document>` for this purpose. Until now, however, this is only inserted manually if needed and not added as a general rule. The XProc 3.0 pipeline needs to take this into account from the outset and simply offer this functionality by adding an invocation parameter.

3.3. Too many tools means too many dependencies

The main problem with the existing batches is the stability and maintainability of the pipelines. That is because the batches had to use different tools for specific tasks, e.g.:

- Beyond Compare for synchronising the XML files and images in the source folder and in the result folders
- 7-Zip to create Zip archives
- curl for transferring results to REST endpoints

The more tools are involved, the greater the dependencies and the greater the risk of breaking changes with future updates of these tools – of which there were quite a few in that dozen years. It also makes it harder to run the batches from different machines. That is because it has to be ensured that each machine provides these specific tools in their specific version. XProc 3.0 offers the necessary functionalities out of the box, which provided the opportunity to significantly reduce dependencies by minimising the tools involved in the pipelines.

4. New requirements for next version

In addition to the pain points mentioned above, there were also some requested improvements.

4.1. Future-proof approach and improved maintainability by adding a separate orchestration layer

As stated above, the existing batches are not only based on XSLT but also call other tools to fulfil different tasks. Using the command line interface of these tools has the disadvantage of a strong connection between a logical task (e.g. create Zip archives) and a specific software implementation (e.g. call 7-Zip with the following parameters). Adapting to a new software version with changes in the command line interface involves changes in the batch. Using other software to fulfil the task is often associated with great costs. Languages such as ANT etc. introduce an extra level of abstraction that models the logical structure of the task and uncouples it from a specific implementation. Using such a level of abstractions makes workflows more robust against changes in the implementing software. Changing the task's implementation can be as easy as changing one configuration file instead of changing every workflow document that uses the task.

4.2. Increased quality through validation of XML sources using T_0 XSD as well as validation of XML results using specific versions of T_0 DTD

While the existing pipelines validate all XML source documents, XML results were not validated. Simply validating the XML results manually during stylesheet development and subsequently applying correct XSLT transformations while relying on valid results was sufficient. The nature of the source-to-result transformation for the XML documents is not one-to-one, but one-to-many – producing up to twelve versions of XML result documents for one source document. With regard to the quality of the results, none of these twelve XML results should appear in the Zip archive if at least one result document is not valid with respect to its specific version of T_0 DTD.

4.3. Increased quality by additional validation of XML results using Schematron

Schematron validations (of XML sources) are performed by Medical Editors and Content Managers during editing in <oXygen /> XML Editor. There is even a batch based pipeline integrated in the content management system to perform a Schematron validation using Skeleton that generats an easy to understand PDF from the SVRL report. Unfortunenatly this Skeleton implementation isn't integrated in the existing batches for the exports. Therefore the XProc 3.0 pipeline should perform a additional Schematron validation when the DTD validation mentioned above is done.

4.4. Summarised, formatted and easily comprehensible log files

The target audience of the content management system is Medical Editors, i.e. non-technical users. For every new edition of their patient education leaflet, they have to perform an export so that an integration test of their leaflet can be done in E-ConsentPro for quality assurance purposes. After moving to XProc 3.0, the pipelines will produce a lot more logging messages, e.g. because of the newly performed additional validations. The existing batches simply stored Saxon’s error messages to different text files. For the non-technical audience, there should be just one single HTML file that serves as a summarised log. There should also be some basic CSS and the use of `<details>` to show/hide additional information as well as the possibility to filter log messages by error category.

4.5. Performance improvement by omitting unnecessary images from the Zip archive

The XML sources contain links to images supplied in the same folder as the XML sources. The improved pipeline has to make sure that all references are valid, i.e. every link goes to an existing image. The batch simply copied all images from the source folder to the result folders. The XProc 3.0 pipeline should be able to copy just the referenced images, but only the images referenced in the XML results, not in the XML sources. That is because there are cases when the pipeline omits the XML result because of special XPath conditions in the source or because the generated XML result is not valid against its specific version of T_0 DTD. Although these additional unnecessary images are subsequently ignored by the REST services, it would be better to just omit them. This would result in smaller Zip archives and therefore faster transmission to the REST service and faster processing.

4.6. Limiting processing to specific sources from the source folder

When a new requirement is developed in the XSLT stylesheets, special test XML sources are created. Normally, the source folder in the file system contains all source XML files, which the new test XML sources are added to. Processing the whole source folder is necessary to avoid regressions. But this takes a couple of hours and, during stylesheet development, only the new test XML sources are relevant. So that the source folder does not have to be changed manually, an easy way to use another source folder and/or a specific file filter is needed. The XProc 3.0 pipeline should be able to use a custom source folder and/or a specific file filter as invocation parameters.

5. New system based on XProc 3.0

Having discussed the original batches and the requirements for the new system, we can now move on to the new pipeline system using XProc 3.0. If we take another look at Figure 3 [48] and Figure 4 [50], it is easy to see that the first one is a good deal more complex than the second one. Our coverage will therefore concentrate primarily on the pipeline replacing the first batch since the basic concepts and strategies for developing the two pipelines are fundamentally the same. The difference between the two pipelines is discussed further below.

The general approach in our move to XProc 3.0 was to replace the double control flow of the original solution with a single control flow in a pipeline. As Figure 3 [48] shows, the original batch incorporated twelve sub-batches, one for each parameter group. Inside each of these batches, an XSLT stylesheet was called to collect and process all documents in the input folder with the given parameters. As we wanted to eliminate as much batch scripting as possible, the first or outer control flow had to be replaced by an XProc 3.0 pipeline. Given the new requirements, we had to remove the inner, XSLT-based control flow as well, thus giving us one iteration over all documents in the input folder instead of twelve iterations (and potentially more in the future).

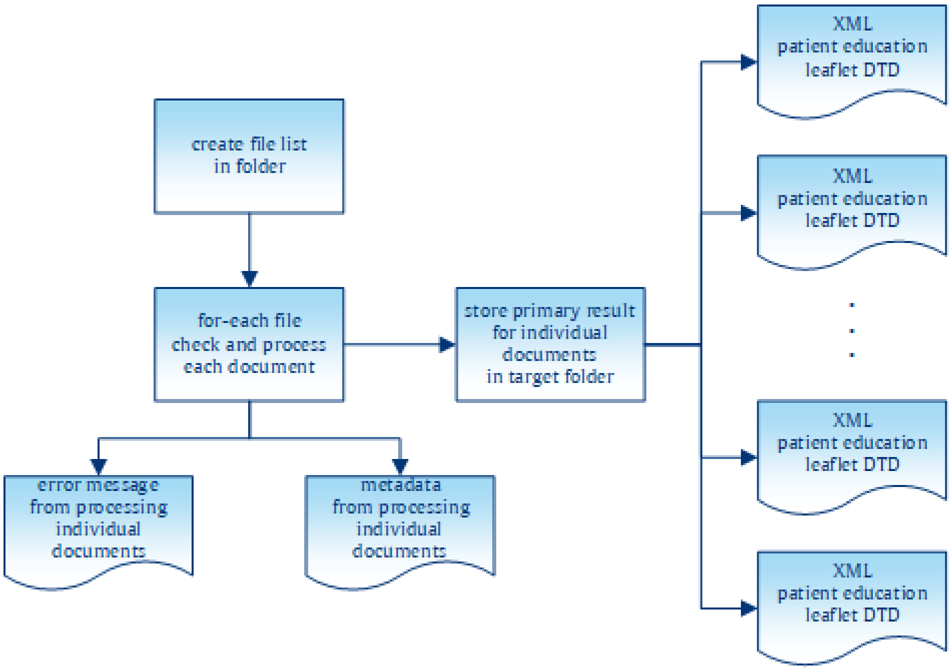
The basic reason for this was the new requirement to check the picture references: A document in the input folder should only be processed if all references to pictures in the document are valid, i.e. point to an existing file. As this is a property of an input document, it makes sense to address this once and for all, and hence start the twelve XSLT transformations only after all picture references have been checked.

Doing away with batch scripting also necessitates two other additions to the pipeline: As we no longer call XSLT via Saxon's command line, we can no longer use its powerful command line interface to perform an `XInclude` and to validate the source documents. As XProc 3.0 has the steps `<p:xinclude>` and `<p:validate-with-xml-schema>`, these two tasks can be easily performed before the stylesheet transformations begin.

The well-established and tested XSLT stylesheets are of course reused in the pipeline. However, this first stylesheet is not called from a batch anymore but via the `<p:xslt>` step instead. Additionally, the XProc pipeline uses a different entry point to the stylesheet than the batch: The latter called a named template that creates a collection of documents in a given folder matching a given pattern of file names. For each document in this collection, a template is then called by matching its root element.

The XProc 3.0 pipeline calls the same template by matching the document's root element, but creates the sequence of the documents to be processed itself: XProc's `<p:directory-list>` produces a document reflecting the content of a given directory, possibly using include and exclude filters. The pipeline then iterates over each `<c:file>` element to subject the respective document to processing.

Figure 5. A bird's eye view of the new system


create file list in folder
for-each file check and process each document
store primary result for individual documents in target folder
error message from processing individual documents
metadata from processing individual documents
XML patient education leaflet DTD
XML patient education leaflet DTD
XML patient education leaflet DTD
XML patient education leaflet DTD

The second change to the existing XSLT stylesheet is to unroll the `@saxon:next-in-chain` concatenation of stylesheets to a sequence of explicit calls of `<p:xslt>`. In Saxon, you can use `@saxon:next-in-chain` inside a `<xsl:output>` to direct the stylesheet's output to another stylesheet. This is a convenient way to chain stylesheets and thus decompose complex processing into a set of smaller stylesheets. This approach helps to improve the quality of code by breaking down a complex task into smaller pieces that are easier to manage. The downside is that `@saxon:next-in-chain` is a Saxon-only extension attribute, which is not supported by other processors and is not guaranteed to be a Saxon feature in the future. With the XPath 3.1 function `fn:transform()`, we have a way to chain stylesheet execution together in a standard-compliant manner. It would therefore be easy to rewrite the existing stylesheets to get rid of `@saxon:next-in-chain` and replace it with a cascade of `fn:transform()`. Hence, there is no need to change technology from XSLT to XProc if you are looking for a standard-compliant way to develop decomposed stylesheets and then chain them together to perform the general transformation.

In our case, using XProc 3.0 provides an additional advantage for developing decomposed stylesheets: With `<p:store>`, XProc 3.0 has a step for storing documents that can easily be used to improve debugging pipelines and and – in our case – decomposed stylesheets. Unlike in XProc 1.0, `<p:store>` is fully transparent, meaning that the document on the input port is stored *and* delivered on the output port. Therefore adding a `<p:store>` (almost) anywhere in your pipeline does not break the “normal” flow of documents, but provides great debugging opportunities. Switching off debugging is also pretty easy in XProc 3.0: The attribute `@use-when` associated with a Boolean expression can be used on (almost) any step in XProc. If the expression is evaluated as `false`, the step (and all its descendants) are effectively excluded from the pipeline. Therefore, by using `<p:store use-when="expr" />`, we can easily switch the generation of debugging information on and off. Since XProc 3.0 also introduced static options that could be used in XPath expressions, switching debugging on and off directly from the pipeline invocation in the command line is the way to go. To sum this point up: Unrolling `@saxon:next-in-chain` concatenation into an XProc pipeline helps to improve the quality of code by splitting larger tasks into smaller pieces.

What we have discussed so far might be considered a general blueprint for embedding complex XSLT stylesheets into an XProc pipeline. Running this pipeline is, apart from the improvements in debugging, largely equivalent to invoking the original stylesheet. Let us now have a look at the advancements added to the overall process with the XProc pipeline. As you can see in Figure 5 [54], the general process can be described in three main parts:

1. Obtaining the URIs of the documents to be processed
2. Processing any individual documents and (possibly) storing the results to disk
3. Collecting the information from the individual processing to
    a. Create the Zip archive
    b. Generate a report about the processing

In the overall design, the `<p:for-each>` step has two purposes: (1) It processes every selected document, i.e. applying the XSLT stylesheets and, if the processing was successful, storing the produced documents to disk. (2) It produces meta information about the document processing: For every processed source document, a report document is created. The latter contains success or error reports for each stage of the source document processing, references to the image files in the source document, and a list of URIs pointing to the created result documents. We will look at these report documents later. For now, let us focus on the first purpose of the block inside the `<p:for-each>` step.

When unrolling the `@saxon:next-in-chain` sequence from the original batch, we first get a sequence of two to six `<p:xslt>` steps, each followed by a `<p:store>` for debugging purposes. The first improvement to be added to the pipeline was the validation of the result document(s). Until then the quality of the result document(s) relied on the validity of the source document and the correctness of the XSLT stylesheet(s) to produce the result documents. One requirement for the XProc pipeline was to add a validation for the result document(s) and store only those documents that prove to be valid. As XProc 3.0 defines steps for validation with RELAX NG, XML Schema and Schematron (as well as NVDL and JSON schemas), this could be achieved quite easily. The result of the last XSLT transformation is connected to the input port of a validation step. If the validation is successful, i.e. the document is valid, it appears on the step's result port and can be stored. If the document proves to be invalid, the validation step raises an error, which is then caught to create an element for the report document.

For the “fragengruppe_2_FHIR-Questionnaire” the validation takes the form of an XML schema and a Schematron document. To validate the produced FHIR document, you simply chain a `<p:validate-with-xml-schema>` and `<p:validate-with-schematron>` after each other to obtain a complete validation of the document. For the other pipeline, called “fragengruppe_2_evidence” the resulting documents are in-house documents for which a DTD is the authoritative grammar. This poses some problems to XProc 3.0 since, as you might have established from the aforementioned list of supported validation languages, there is no step for DTD validation included. With `<p:load>`, however, pipeline authors can ask for a DTD validation of the document to be loaded. As with the “normal” validation steps, this either returns a valid document or an error is thrown. Unlike with the normal validation steps, however, the document to be validated does not appear on the input port but has to be loaded from a secondary store via URI. To circumvent this restriction, we switched the steps around: Instead of validating the (in-memory) document first and then storing if the validation succeeds, we first store the document and then validate it via reloading. As a consequence, invalid artifacts are written to disk, which does not occur in the other pipelines where we can validate before the document is stored. This was a deliberate decision, because it would be easy to delete the stored document from disk once we have discovered its invalidity.

Another aspect of the quality improvements for the produced results is related to image referencing. Along with the source XML documents, the source folder also contains image files. Some of the image files are referenced in the documents, but not all. On the other hand, documents might have a reference to a non-existing image file. To improve the overall quality of the results, the XProc pipeline was required to check the source documents for image references and to raise an error if an image file is referenced that is not present in the source folder. In addition, the pipeline has to keep track of the referenced image files, so that only referenced images are included in the resulting Zip archive.

With XProc 3.0, both requirements are pretty easy to fulfill: The pipeline iterates over all image references in the selected source documents and obtains their URI. With the `<p:file-info>` step, you can then obtain information about the referenced image: If the referenced resource exists, a `<c:file>` document appears on the `result` port. If the document does not exist, a `<c:error>` document will appear on this port. In the first case, an entry for the archive manifest is generated, In the second case, an error report element is created to flag the error for the report to be generated.

The outlined algorithm solves both shortcomings in the existing batch system: Since only images with a corresponding `<c:entry>` element are included in the Zip archive, the latter will only contain images actually referenced in at least one source document. And since every image reference in a source document is tested, the resulting report will include every error for invalid image references.

However, some post-processing of the image-related `<c:entry>` elements proved to be necessary: For efficiency reasons, the image references were tested for the source documents, but not for the produced documents. This is possible since the XSLT transformations do not add any image references. In general, there will be a 1:m relationship between the source documents and resulting documents, but since the produced documents are validated, there may be a 1:0 relationship if the validation for all produced documents fails. In this special case, there might be a `<c:entry>` element for an image document that is not referenced by any resulting document. Before actually generating the Zip archive, some cleanup has to take place. Every entry for an image documents records the URI of the source document it was derived from as well as any resulting XML document. For every image-related `<c:entry>`, the cleanup has to check whether there is a corresponding `<c:entry>` from a produced document. If not, the image-related entry is simply deleted, meaning that the resulting archive will only contain images actually referenced in a contained XML document.

The last aspect of the new XProc pipeline system to mention here is the improvement of logging or reporting. One new feature of XProc 3.0 that very much supports this requirement is the addition of `<p:catch>` with specified error codes. In XProc 1.0, we just had a `<p:try>`/`<p:catch>` where every error raised in the `<p:try>` block had to be handled in a single `<p:catch>` block. The ability to write different `<p:catch>` blocks for specific errors results in a more readable pipeline. This is also facilitated by the definition of more fine-grained error conditions for each individual step in XProc 3.0. This makes it very easy to identify what exactly went wrong inside an XProc pipeline, and to react to or report on the exact problems that occurred.

Our pipeline makes extensive use of `<p:try>`/`<p:catch>` to improve reporting, which is an important requirement as stated above. If you have a look at Figure 5 [54] again, the pipeline does not only produce XML documents representing the transformation results (shown on the right side of the code block), but also reports about the pipeline process (failing out at the bottom of the code block). For (almost) every document found in the source directory, the pipeline produces a report document. Here is an excerpt of such a document:

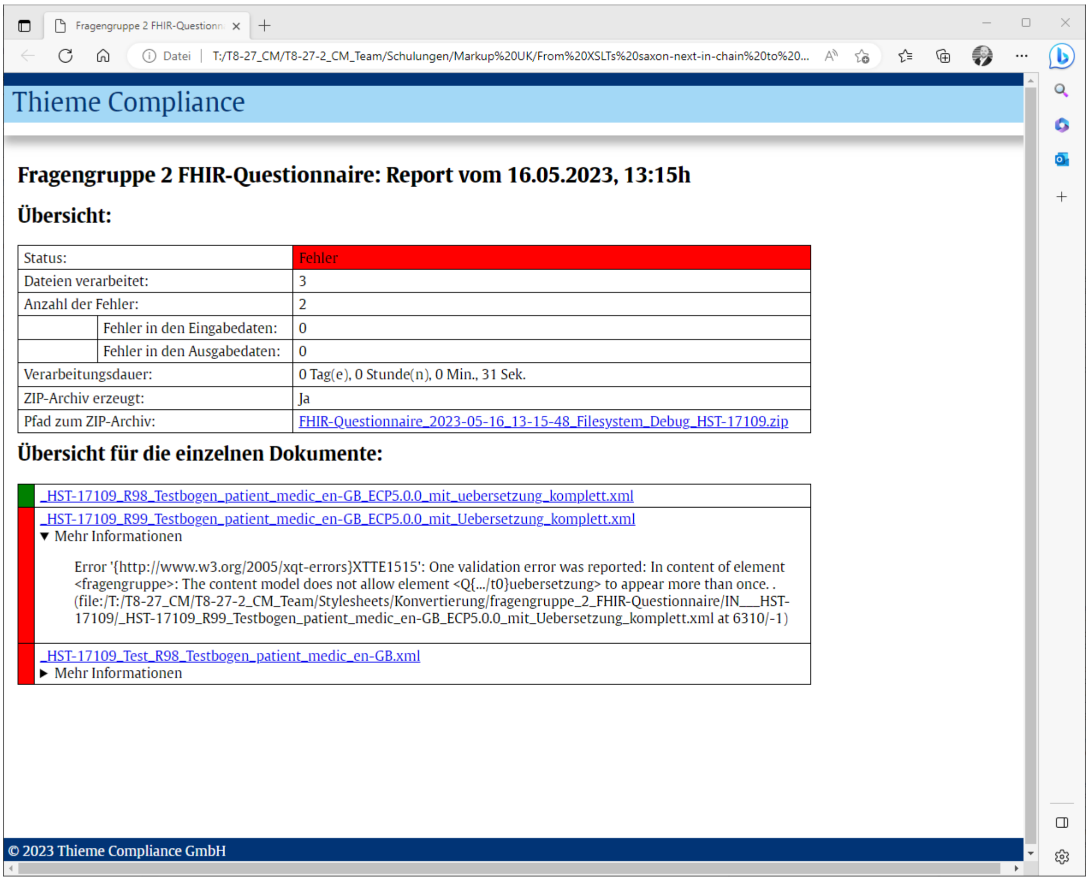
```
<tcg:report file="reference-to-source-doc">
   <tcg:report-done phase="validation">
      <successfully-validated />
   </tcg:report-done>
   <tcg:report-done phase="processFileRefs">
      <c:entry name="name-of-zip-entry-for-pic1"
               href="path-to-pic1"
               found-in="reference-to-source-doc" />
   </tcg:report-done>
   <tcg:report-done phase="2.5.1">
     <c:entry name="name-of-zip-entry-for-doc1"
               href="path-to-doc1"
               derived-from="reference-to-source-doc" />
   </tcg:report-done>
   <tcg:report-error phase="2.6.0">
     <c:errors><!-- detailed error report here --></c:errors>
   </tcg:report-error>
</tcg:report>
```

This snippet shows a processing report for a source document the URI of which is reported in the `@file` attribute. The document passes the source validation successfully, and it contains a valid reference to an image file. It is transformed in phase `"2.5.1"`, thus producing a result stored at the recorded position. However, the transformation in phase `"2.6.0"` was not successful, as a result of which a `<tcg:report-error>` is included to flag the error and to report corresponding details. The `<c:entry>` elements are used to create the Zip archive. The attributes `@found-in` and `@derived-from` are used to make sure that an image file is included in the archive only if the source document containing the image references also passes transformation.

Based on the collected processing reports, the last two tasks of the pipeline are performed: creating the Zip archive and then creating the final report. Creating the archive is pretty straightforward. XProc 3.0's `<p:archive>` step has an input port `manifest`. The expected document has a `<c:manifest>` root element with `<c:entry>` elements denoting the expected archive entries. As we already created `<c:entry>` elements in our document reports, we just need to extract those elements from the report, wrap it in the expected element root and then call `<p:archive>` to create the Zip archive.

The final report is also based on the collected report elements: The reports contains information about every single processed XML document in the source folder, to which we add aggregated data such as the number of processed files, the number of detected errors or the complete processing duration. Finally, this report is processed with an XSLT stylesheet to create an HTML document and thus provide the improved logs requested in the initial requirements.

Figure 6. Summarised HTML log


Fragengruppe 2 FHIR-Questionn
Datei
T:/T8-27_CM/T8-27-2_CM_Team/Schulungen/Markup%20UK/From%20XSLTs%20saxon-next-in-chain%20to%20...

Thieme Compliance

Fragengruppe 2 FHIR-Questionnaire: Report vom 16.05.2023, 13:15h

Übersicht:

Status:		Fehler
Dateien verarbeitet:		3
Anzahl der Fehler:		2
	Fehler in den Eingabedaten:	0
	Fehler in den Ausgabedaten:	0
Verarbeitungsdauer:		0 Tag(e), 0 Stunde(n), 0 Min., 31 Sek.
ZIP-Archiv erzeugt:		Ja
Pfad zum ZIP-Archiv:		FHIR-Questionnaire_2023-05-16_13-15-48_Filesystem_Debug_HST-17109.zip

Übersicht für die einzelnen Dokumente:

🟩	_HST-17109_R98_Testbogen_patient_medic_en-GB_ECP5.0.0_mit_uebersetzung_komplett.xml
🟥	_HST-17109_R99_Testbogen_patient_medic_en-GB_ECP5.0.0_mit_Uebersetzung_komplett.xml ▼ Mehr Informationen Error '{http://www.w3.org/2005/xqt-errors}XTTE1515': One validation error was reported: In content of element <fragengruppe>: The content model does not allow element <Q{.../t0}uebersetzung> to appear more than once. . (file:/T:/T8-27_CM/T8-27-2_CM_Team/Stylesheets/Konvertierung/fragengruppe_2_FHIR-Questionnaire/IN___HST-17109/_HST-17109_R99_Testbogen_patient_medic_en-GB_ECP5.0.0_mit_Uebersetzung_komplett.xml at 6310/-1)
🟥	_HST-17109_Test_R98_Testbogen_patient_medic_en-GB.xml ▶ Mehr Informationen

© 2023 Thieme Compliance GmbH

As stated above, since the two batches were similar in their basic concepts, the two pipelines will also resemble each other. There are two basic differences to the pipeline already discussed:

1. While the first pipeline is only a cascade of two XSLT stylesheet, the FHIR has a total of five XSLTs. The implementation is pretty straightforward as the result of the first `<p:xslt>` step serves as the input for the next one. As a consequence, instead of two consecutive calls to `<p:xslt>`, we have a sequence of five interconnected calls in the FHIR pipeline.

2. The second, more conceptual difference is the way that the pipeline results are validated. While the pipeline discussed above uses a DTD validation, FHIR documents employ a different concept. The standard defines an XML schema and a Schematron schema to validate the documents. As both of these validation technologies are supported in XProc 3.0 by individual steps, incorporating FHIR validation simply requires a pair of `<p:validate-with-xml-schema>` and `<p:validate-with-schematron>` with the produced document on the default input port.

6. Takeaways

What are the takeaways from our project?

6.1. Smooth transition to XProc 3.0

First of all, we have to say that the move from batch scripts to XProc 3.0 was a very smooth experience. Compared to developing pipelines with XProc 1.0, the process is a lot faster. This is thanks both to the wealth of syntactic sugar and to the much cleaner concept of some frequently used steps. The type system for variables and options provides a new level of security. The availability of XPath 3.0 and the respective functions improves programming a lot. And we are pleased to say that we did not miss parameter ports for a second.

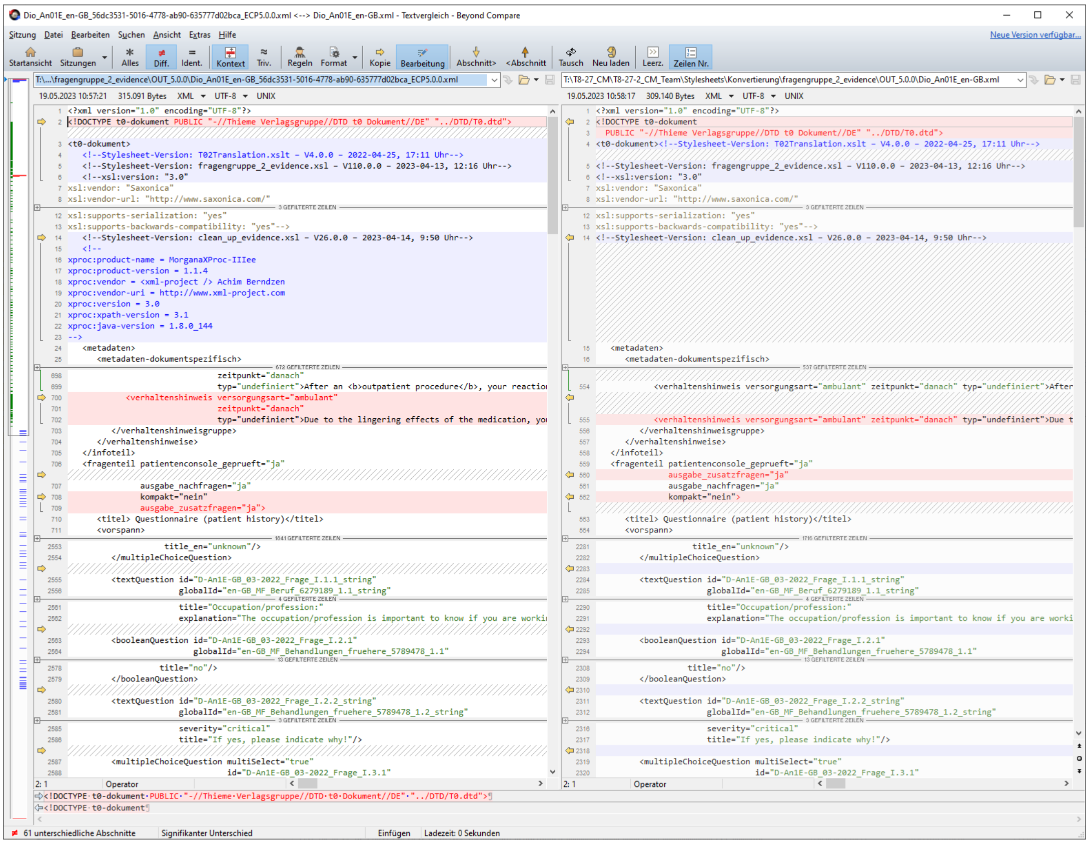
6.2. MorganaXProc-IIIse worked well and could even be improved over the course of the project

MorganaXProc-IIIse turned out to be a reasonable tool. The complete task could be fulfilled without any custom additions. Having to develop a complex pipeline system with a lot of documents to process helped a lot in improving MorganaXProc. During pipeline development, some bugs were found in the software and most of them have already been fixed. Additionally, we identified pain points for optimisation from using MorganaXProc in a large real-world project, which will be reflected in new features to be added in the future.

6.3. Serialisation is now done by MorganaXProc and no longer by Saxon

One thing to note is that with XProc 3.0 the serialisation is done by MorganaXProc even though the transformation is still performed by Saxon. Since the XSLT transformation is now orchestrated by XProc 3.0, a `<xsl:output>` declaration in the stylesheet is now ignored. In particular, the serialisation differs with regard to the order of attributes and whether or not they are presented in a new line. In addition, Also Saxon provides powerful serialisation features which MorganaXProc currently lacks. For instance, there is `@saxon:line-length`, `@saxon:attribute-order` or `@saxon:double-space`, which were used with the batch pipelines. These are useful features for (further) increasing the human readibility of the XML result documents.

Figure 7. Serialisation done by MorganaXProc (left) and Saxon (right)



Dio_An01E_en-GB_56dc3531-5016-4778-ab90-635777d02bca_ECP5.0.0.xml <--> Dio_An01E_en-GB.xml - Textvergleich - Beyond Compare

T:\...\fragengruppe_2_evidence\OUT_5.0.0\Dio_An01E_en-GB_56dc3531-5016-4778-ab90-635777d02bca_ECP5.0.0.xml
19.05.2023 10:57:21 315.091 Bytes XML UTF-8 UNIX

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE t0-dokument PUBLIC "-//Thieme Verlagsgruppe//DTD t0 Dokument//DE" "../DTD/T0.dtd">
<t0-dokument>
   <!--Stylesheet-Version: T02Translation.xslt – V4.0.0 – 2022-04-25, 17:11 Uhr-->
   <!--Stylesheet-Version: fragengruppe_2_evidence.xsl – V110.0.0 – 2023-04-13, 12:16 Uhr-->
   <!--xsl:version: "3.0"
xsl:vendor: "Saxonica"
xsl:vendor-url: "http://www.saxonica.com/"
                              3 GEFILTERTE ZEILEN
xsl:supports-serialization: "yes"
xsl:supports-backwards-compatibility: "yes"-->
   <!--Stylesheet-Version: clean_up_evidence.xsl – V26.0.0 – 2023-04-14, 9:50 Uhr-->
   <!--
xproc:product-name = MorganaXProc-IIIee
xproc:product-version = 1.1.4
xproc:vendor = <xml-project /> Achim Berndzen
xproc:vendor-uri = http://www.xml-project.com
xproc:version = 3.0
xproc:xpath-version = 3.1
xproc:java-version = 1.8.0_144
-->
   <metadaten>
      <metadaten-dokumentspezifisch>
                              672 GEFILTERTE ZEILEN
                              zeitpunkt="danach"
                              typ="undefiniert">After an <b>outpatient procedure</b>, your reactio
            <verhaltenshinweis versorgungsart="ambulant"
                              zeitpunkt="danach"
                              typ="undefiniert">Due to the lingering effects of the medication, yo
         </verhaltenshinweisgruppe>
      </verhaltenshinweise>
   </infoteil>
   <fragenteil patientenconsole_geprueft="ja"
               ausgabe_nachfragen="ja"
               kompakt="nein"
               ausgabe_zusatzfragen="ja">
      <titel> Questionnaire (patient history)</titel>
      <vorspann>
                              1841 GEFILTERTE ZEILEN
                     title_en="unknown"/>
         </multipleChoiceQuestion>
         <textQuestion id="D-An1E-GB_03-2022_Frage_I.1.1_string"
                       globalId="en-GB_MF_Beruf_6279189_1.1_string"
                              4 GEFILTERTE ZEILEN
                       title="Occupation/profession:"
                       explanation="The occupation/profession is important to know if you are worki
         <booleanQuestion id="D-An1E-GB_03-2022_Frage_I.2.1"
                          globalId="en-GB_MF_Behandlungen_fruehere_5789478_1.1"
                              13 GEFILTERTE ZEILEN
                    title="no"/>
         </booleanQuestion>
         <textQuestion id="D-An1E-GB_03-2022_Frage_I.2.2_string"
                       globalId="en-GB_MF_Behandlungen_fruehere_5789478_1.2_string"
                              3 GEFILTERTE ZEILEN
                       severity="critical"
                       title="If yes, please indicate why!"/>
         <multipleChoiceQuestion multiSelect="true"
                                 id="D-An1E-GB_03-2022_Frage_I.3.1"
```

2: 1

T:\T8-27_CM\T8-27-2_CM_Team\Stylesheets\Konvertierung\fragengruppe_2_evidence\OUT_5.0.0\Dio_An01E_en-GB.xml
19.05.2023 10:58:17 309.140 Bytes XML UTF-8 UNIX

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE t0-dokument
  PUBLIC "-//Thieme Verlagsgruppe//DTD t0 Dokument//DE" "../DTD/T0.dtd">
<t0-dokument><!--Stylesheet-Version: T02Translation.xslt – V4.0.0 – 2022-04-25, 17:11 Uhr-->
<!--Stylesheet-Version: fragengruppe_2_evidence.xsl – V110.0.0 – 2023-04-13, 12:16 Uhr-->
<!--xsl:version: "3.0"
xsl:vendor: "Saxonica"
xsl:vendor-url: "http://www.saxonica.com/"
                              3 GEFILTERTE ZEILEN
xsl:supports-serialization: "yes"
xsl:supports-backwards-compatibility: "yes"-->
<!--Stylesheet-Version: clean_up_evidence.xsl – V26.0.0 – 2023-04-14, 9:50 Uhr-->
   <metadaten>
      <metadaten-dokumentspezifisch>
                              537 GEFILTERTE ZEILEN
            <verhaltenshinweis versorgungsart="ambulant" zeitpunkt="danach" typ="undefiniert">After
            <verhaltenshinweis versorgungsart="ambulant" zeitpunkt="danach" typ="undefiniert">Due t
         </verhaltenshinweisgruppe>
      </verhaltenshinweise>
   </infoteil>
   <fragenteil patientenconsole_geprueft="ja"
               ausgabe_zusatzfragen="ja"
               ausgabe_nachfragen="ja"
               kompakt="nein">
      <titel> Questionnaire (patient history)</titel>
      <vorspann>
                              1716 GEFILTERTE ZEILEN
                     title_en="unknown"/>
         </multipleChoiceQuestion>

         <textQuestion id="D-An1E-GB_03-2022_Frage_I.1.1_string"
                       globalId="en-GB_MF_Beruf_6279189_1.1_string"
                              4 GEFILTERTE ZEILEN
                       title="Occupation/profession:"
                       explanation="The occupation/profession is important to know if you are worki

         <booleanQuestion id="D-An1E-GB_03-2022_Frage_I.2.1"
                          globalId="en-GB_MF_Behandlungen_fruehere_5789478_1.1"
                              13 GEFILTERTE ZEILEN
                    title="no"/>
         </booleanQuestion>

         <textQuestion id="D-An1E-GB_03-2022_Frage_I.2.2_string"
                       globalId="en-GB_MF_Behandlungen_fruehere_5789478_1.2_string"
                              3 GEFILTERTE ZEILEN
                       severity="critical"
                       title="If yes, please indicate why!"/>

         <multipleChoiceQuestion multiSelect="true"
                                 id="D-An1E-GB_03-2022_Frage_I.3.1"
```

2: 1

```
<!DOCTYPE·t0-dokument·PUBLIC·"-//Thieme·Verlagsgruppe//DTD·t0·Dokument//DE"·"../DTD/T0.dtd">¶
<!DOCTYPE·t0-dokument¶
```

61 unterschiedliche Abschnitte   Signifikanter Unterschied   Einfügen   Ladezeit: 0 Sekunden

We had to ensure that the XML results of the XProc 3.0 pipelines are similar to the results of the batch pipelines. But there were two main issues:

1. There is some siginificant whitespace handling in the XSLT stylesheets.
2. Some transformations are now done in XProc itself using `<p:insert>` and no longer within the XSLT stylesheets.

We were not aware of the serialisation behaviour at first and, aesthetic considerations aside, these shortcomings in MorganaXProc-IIIse made it difficult to compare the results of the original batches with the pipeline results. Comparing the XML results therefore took more time, and the XMLs had to be pretty-printed first, with the downside that this falsified the whitespace handling.

6.4. Performance problems with FHIR XML schema

The low performance of the FHIR pipeline gave us some headaches. It was significantly slower than the other pipeline although being very similar in the its structure. Turned out that the result validations with the official FHIR schema was the culprit. Initially we used “`fhir-all`” in `<p:validate-with-xml-schema>`, but that was extremely slow. Turns out that schema document just imported about 140 other schema documents, and each of them importing the same schema. Luckily there is an other official schema document “`fhir-single`” which contains a complete schema document. Using this schema document did the trick and we are now happy with the pipeline's performance.

6.5. XProc pipeline optimisation by loading stylesheets only once at the beginning

The pipeline development was not without drawbacks: The first (and natural) approach to processing the documents in the input folder would be:

```
<p:for-each>
    <p:xslt>
        <p:with-input port="stylesheet" href="the-stylesheet.xsl" />
    </p:xslt>
</p:for-each>
```

However, if you process around 15,000 source documents, the stylesheet document is loaded each time too. Since this is completely inefficient, we changed it to:

```
<p:load href="the-stylesheet.xsl" name="stylesheet" />
<!-- ... -->
<p:for-each>
    <p:xslt>
        <p:with-input port="stylesheet" pipe="@stylesheet" />
    </p:xslt>
</p:for-each>
```

This is a bit better performance-wise, but makes the pipeline more difficult to read. Moreover, it only partially resolves the inefficiency issue because the stylesheet has to be compiled every time the `<p:for-each>` is executed. The same applies to the XML schemas or the Schematron. They have to be prepared every time to be usable, although they do not change in our case. Using an XML catalog does not help here, because it only caches the document. The most effective solution would be to cache the ready-to-use stylesheet etc. But simply caching them as default processor behaviour is not feasible in XProc 3.0. Looking at the specifications, it is perfectly possible to rewrite documents or stylesheets etc. within `<p:for-each>` so that a later iteration depends on documents produced in an earlier one.

A processor could perform some optimisation here by checking whether a certain document is written inside a <p:for-each>. However, since catalog resolution etc. frequently takes place in XProc, there is no simple and reliable way to do this. From our perspective, some more investigation of this problem is necessary: One solution could be an (extension) attribute on <p:with-input> allowing a pipeline author to declare a stylesheet, schema etc. to be cacheable. Whether this is implemented in XProc at language level or takes the form of a vendor-specific extension is a discussion for the future.

6.6. Feature request for XProc: please add <p:validate-with-dtd>

XProc 3.0 offers a whole range of possibilities for validating documents using:

- <p:validate-with-json-schema>
- <p:validate-with-nvdl>
- <p:validate-with-relax-ng>
- <p:validate-with-schematron>
- <p:validate-with-xml-schema>

However, there is no equivalent for validating XML documents with DTD.

Even though XML Schema became a W3C recommendation already in 2001, and RELAX NG was also defined then, there are still legacy systems that only support DTD. In rare cases, it is possible to enhance these systems by switching to XML Schema, but this is generally not an option and these systems have to be supported nevertheless.

The lack of DTD validation in XProc 3.0 means that the pipelines have to use a workaround. The XML result must first be stored to the file system and then loaded immediately from there again using <p:load>. This is because, surprisingly, DTD validation is possible with XProc 3.0 but only while loading a document. Finally, the previously stored and validated XML result has to be deleted, because its no longer needed. While this workaround does its job, the ability to do this in memory using something like <p:validate-with-dtd> would be much more effective.

There is perhaps an argument for adding DTD validation to the pipeline using <p:declare-step>. However, this requires the XProc processor to realise that only a memory stream is needed, causing it to perform an optimisation and avoid storing the result to the file system and reloading it. Even if this would work, anybody who wants to support DTD has to add the following step to their pipelines instead of just using <p:validate-with-dtd>.

```
<p:declare-step type="tcg:validate-with-dtd">
    <p:input port="source" sequence="false" content-types="xml" />
    <p:output port="result" sequence="true" content-types="xml" />

    <p:pipe step="load" />

    <p:store href="foo.xml" name="store" />

    <p:load href="foo.xml" name="load" parameters="map{'dtd-validate' : true()}" depends-on="store" />

    <p:file-delete href="foo.xml" depends-on="load" />
</p:declare-step>
```

So please, XProc working group, support DTD validation natively and add <p:validate-with-dtd> and Bob's your uncle.

Bibliography

[Siegel:2019] Erik Siegel. *An introduction to XProc 3.0*. XML.com. https://www.xml.com/articles/2019/11/05/introduction-xproc-30/. 2019.

[Siegel:2020] Erik Siegel. *XProc 3.0. Programmer Reference*. XML Press. Laguna Hills, CA. 2020.

[Siegel:2020a] Erik Siegel. *XProc 3.0 - Connecting steps using ports*. XML.com. https://www.xml.com/articles/2020/01/23/xproc-30-connecting-steps-using-ports/. 2020.

[Siegel:2020b] Erik Siegel. *XProc 3.0 - Strategies for merging documents*. XML.com. https://www.xml.com/articles/2020/11/16/xproc-30-strategies-merging-documents/. 2020.

[7-Zip] is a file archiver with a high compression ratio. https://www.7-zip.org

[Beyond Compare] is a data comparison program developed by Scooter Software. In addition to comparing files, the program can also compare entire directories as well as FTP directories and archives. The program's particular strengths include its scripting capability and its usage of built-in and extendable comparison rules for different file formats. https://www.scootersoftware.com

[CLS] Content Lifecycle System (CLS) is a component content management system from Empolis Solutions GmbH. https://www.empolis.com/

[curl] is used in command lines or scripts to transfer data. https://curl.se

[E-ConsentPro] is a software solution for digital patient education developed by Thieme Compliance GmbH https://thieme-compliance.de/en/products/e-consentpro-software/

[HL7 FHIR] The Fast Healthcare Interoperability Resources (FHIR, pronounced “fire”) standard is a set of rules and specifications for exchanging electronic healthcare data. It was created by the Health Level Seven International (HL7) healthcare standards organisation. https://www.hl7.org/fhir

[LHC-Forms] is a widget that renders input forms for web-based applications based on FHIR questionnaires provided by the National Library of Medicine. https://lhcforms.nlm.nih.gov/

[LOINC] (Logical Observation Identifiers Names and Codes) is an international system published by the Regenstrief Institute for the unique identification and coding of medical observations, especially laboratory tests. https://loinc.org

[SNOMED CT] (Systematized Nomenclature of Medicine and Clinical Terms) is currently the most comprehensive health terminology in the world, a steadily growing ontology of preferred terms and their synonyms. It is maintained and distributed by SNOMED International. https://www.snomed.org

[XProc] is an XML based programming language for processing documents in pipelines, which involves chaining conversions and other steps together to achieve the desired results. The current version is 3.0. https://xproc.org/

Markup UK

Bridging the Gaps Between XML and TEX

An approach to convert from arbitrary XML to LaTeX by utilizing an XSLT/XProc-based framework.

Martin Kraetke, le-tex publishing services GmbH

There are established software solutions to convert from XML to TeX. Nevertheless, these approaches either limit the configurability of the TeX output or introduce a lot of programming effort. In this paper, we will discuss these methods and present an alternative approach that hopes to offer the greatest possible flexibility based on fixed conversions, an extensible configuration and the injection of XSLT, and that forms the basis for our Open Source typesetting system xerif.

1. Acknowledgements

Patrick Schulz for patiently explaining all the peculiarities of TeX. Gerrit Imsieke for his careful review and his very appreciated suggestions.

2. Introduction

TeX is a powerful typesetting system and popular in scientific publishing due to the superior typesetting of mathematical formulae. The software is published under a permissive Open Source license and offers a rich ecosystem of packages that extend its functionality. LaTeX, a macro system built on top of TeX, provides descriptive commands for common layout elements and simplifies the use of TeX. Although XML and LaTeX are both markup languages, there are several limitations that make it difficult for converting XML to LaTeX. This paper describes common problems, presents existing software solutions and introduces a declarative approach for transforming XML to LaTeX.

3. The Gaps Between XML and LaTeX

XML and LaTeX were designed to serve different purposes but the most obvious difference is the syntax. TeX commands usually start with backslashes, may contain optional arguments in square brackets and mandatory arguments in curly braces. A LaTeX document starts with a preamble that includes the document class, metadata and references to packages. The document body is placed between \begin{document} and \end{document}. A minimal example of a LaTeX document is shown below:

```
\documentclass{article}
\usepackage[british]{babel}
\title{my Markup UK paper}
\author{Martin Kraetke}
\date{\today}
\begin{document}
```

```
\maketitle
\section{Introduction}
This is a paragraph with \textit{italicized text}.
\end{document}
```

While the principles of this syntax are consistent, LaTeX has two main modes with syntactical differences: Text mode is used for regular text and math mode is used for mathematical expressions.

1. In text mode, special characters such as backslash, curly braces, and percent signs have a specific meaning and if they are part of the regular text, they need to be escaped with the backslash character. There are various LaTeX macros to markup the text. For example, \textsubscript{…} and ^{…} are used for sub- and superscripted text.

2. Math mode has a slightly different syntax. Math mode can be entered using either LaTeX environments like \begin{equation}…\end{equation} or dollar signs ($ or $$). Furthermore, math mode introduces new commands and special characters for mathematical expressions. In contrast to text mode, the underscore symbol (_) indicates a subscript and the caret symbol denotes a superscript.

Another difference between XML and TeX is Unicode-compatibility. LaTeX was developed before Unicode and had limited support for non-ASCII characters. Support for Unicode in LaTeX depends on the TeX rendering engines being used. The traditional pdfTeX engine is not Unicode-compatible by default. The input encoding needs to be set correctly and additional font packages need to be used because the original default font encoding (OT1) of TeX was 7-bit and used fonts that only had 128 glyphs. LuaTeX and XeTeX are newer TeX engines and both designed to work with Unicode and OpenType fonts. In summary, when you convert XML to TeX, you need to know which TeX engine is used.

While XML has namespaces to add new semantics, LaTeX allows you to include additional functionality through document classes, packages and custom commands or macros. These mechanisms can extend the capabilities of LaTeX by adding new features or modify existing ones. The sheer number of available packages is one of the major advantages of LaTeX, but can become also a challenge. Certain packages have different approaches for the same problem which results sometimes in an overlapping feature set: For instance, if you just want to underline text, you can do this in several ways. As alternative to the built-in macro \underline{...}, the soul package provides \ul{…} and the ulem package has \uline{…} for the same purpose. You can also define custom macros with \newcommand{\name}{definition}. The fact that TeX/LaTeX is a programming language in its own right and not just a pure markup language makes processing LaTeX files with other programming languages sometimes a challenge. Parsing XML and converting it to LaTeX is definitely easier than parsing arbitrary LaTeX and converting it to XML.

Even though there are a number of macros that are built-in LaTeX and can be considered standard, the choice of macros depends on the available package, the use case and personal preferences. This aspect must be taken into account when implementing an XML to TeX transformation.

Many XML grammars include the CALS or the HTML table model, or offer different schema variants with these models. In LaTeX, tables are a feature that is provided by external packages like tabularx[1]. Here is a brief example:

```
\begin{tabularx}{|c|c|}
Firstname &amp; Surname \\
```

[1] David Carlisle (2020): The tabularx package. Available at http://mirrors.ctan.org/macros/latex/required/tools/tabularx.pdf [http://mirrors.ctan.org/macros/latex/required/tools/tabularx.pdf] (Accessed: May 16, 2023)

```
\hline
Geert &amp; Bormans \\
Ari &amp; Nordström \\
Andrew &amp; Sales \\
Rebecca &amp; Shoob \\
\end{tabularx}
```

This code creates a table with four rows and two columns. The first argument of tabular provides the column declaration. The letter “c” creates a centered column and the pipe (|) symbol indicates a vertical line. Table cells are separated with an ampersand symbol (&) and each row ends with a double backslash (\\). \hline is used to indicate a horizontal line.

If you want to insert a cell that spans across several columns, you can use the \multicolumn instruction. Despite the fact that you are able to adjust the column span with tabularx, the functionality to have row spans as well is not included. Therefore, you need to add another package called multirow[2]. Even though you can use multirow together with tabularx, it seems a bit odd to have different packages for a feature that the author would consider as basic functionality. Of course, you need additional packages for colored borders and cells, when you want to wrap tables over several pages or rotate a table.

4. Methods to Convert XML to LaTeX

4.1. xmltex

xmltex is a simple, non-validating XML parser implemented in TeX by David Carlisle that allows to use LaTeX's typesetting capabilities not just for LaTeX documents but for XML documents as well. xmltex can associate TeX code with XML elements, attributes, processing instructions, and entities as well. However, xmltex can neither validate the XML document with a DTD nor resolve external DTD entities but is able to process local entity declarations.[3]

Because xmltex is written in TeX, you need TeX to invoke xmltex with LaTeX. Here is an example that loads doc.xml with xmltex from a TeX document.

```
\def\xmlfile{doc.xml} % xml file
\input xmltex.tex % load xmltex
```

The parser supports XML namespaces to some extent and can be configured for arbitrary XML, for example a TEI document that contains MathML. For this purpose, namespaces can be defined within a separate macro file, usually with an xmt extension[4]. For each XML document type, a separate xmt file is required. Whenever xmltex processes an XML element with a particular namespace, it loads the corresponding xmt file. The following command can be added to a catalogue configuration (cfg extension) to associate a namespace with a particular xmt file:

```
\NAMESPACE{URL}{xmt-file}
```

The xmltex catalogue configuration allows to associate XML contexts with TeX instructions. For example, you can specify with

```
\XMLelement{element-qname}{attribute-spec} {begin-code}{end-code}
```

[2] Pieter van Oostrum et al. (2021) The multirow, bigstrut and bigdelim packages. Available at http://mirrors.ctan.org/macros/latex/contrib/multirow/multirow.pdf (Accessed: May 15, 2023)

[3] David Carlisle (2000) xmltex: A non validating (and not 100% conforming) namespace aware XML parser implemented in TeX. Available at https://ctan.space-pro.be/tex-archive/macros/xmltex/base/manual.html (Accessed: May 24, 2023)

[4] The null namespace, the XML namespace (http://www.w3.org/1998/xml) and the xmltex namespace (http://www.dcarlisle.demon.co.uk/xmltex) are predeclared.

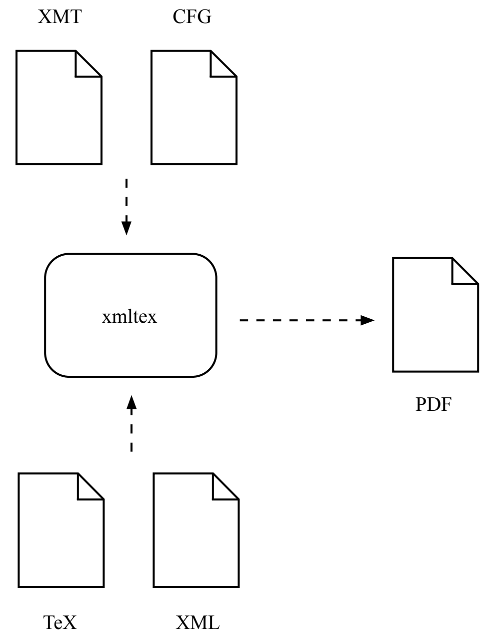
a TeX command for an element. Whenever xmltex encounters the element with this name, the begin and end code are inserted. A minimal xmltex catalogue configuration for TEI is shown in the code listing below.

```
\DeclareNamespace{tei}{http://www.tei-c.org/ns/1.0}
\XMLelement{tei:TEI}
{}
  {\documentclass{article}
     \begin{document}
  }
  {\end{document}}
\XMLelement{tei:teiHeader}
{}
  {}{}
\XMLelement{tei:title}
{}
  {\xmlgrab}
  {\title{#1}
     \maketitle}
\XMLelement{tei:p}
{}
{\par}
  {\par}
```

An xmltex project would usually include a TeX file, a configuration, xmt files for each namespace, and the input XML file. Below is what a typical xmltex project directory structure would look like:

```
MyProject/
  |--main.tex
  |--main.cfg (xmltex-configuration)
  |--doc.xml (XML input)
  |--tei.xmt (xmltex-mapping for TEI)
```

Figure 1. xmltex inputs and outputs



To configure xmltex for an XML schema like TEI, you need to know the nesting depth of a <head/> element to determine whether it is mapped to \chapter{} or \section{}. The xmltex syntax only provides TeX statements that can be mapped to specific XML node names, but does not take into account their actual position in the XML. To address this issue with xmltex, you need to introduce a counter that counts the number of ancestor <div> elements:

```
\newcount\div@counter \div@counter=0
\XMLelement{tei:head}
  {}
  {\xmlgrab}
  {%
\ifnum\div@counter=1\relax
  \chapter{#1}%
\else
  \ifnum\div@counter=2\relax
    \section{#1}%
  \fi
\fi
}
\XMLelement{tei:dvi}
  {}
  {\global\advance\div@counter1}
  {\global\advance\div@counter-1}
```

xmltex does not offer a powerful query language like XPath, on the contrary, you can only associate plain element and attribute names to TeX instructions. More complex context queries represent a programming task that can prove very daunting, given TeX’s macro-expanding processing model. The code is less declarative and hard to maintain.

If this method is getting too complicated, xmtex allows also to modify the output by placing TeX commands directly into the XML source. The xmltex documentation suggests to use elements either with the xmltex namespace or a custom namespace to inject TeX instructions[5]. Another mechanism is provided by using xmltex processing instructions:

```
<?xmltex TeX commands ?>
```

Furthermore, error reporting is virtually nonexistent and various constraints on XML are not enforced. For example, you can configure element names with characters that are not allowed in XML. Another problem is that xmltex isn't actively maintained. The code was last modified in 2000 and moved to GitHub in 2012 without any notable changes except the initial commit[6].

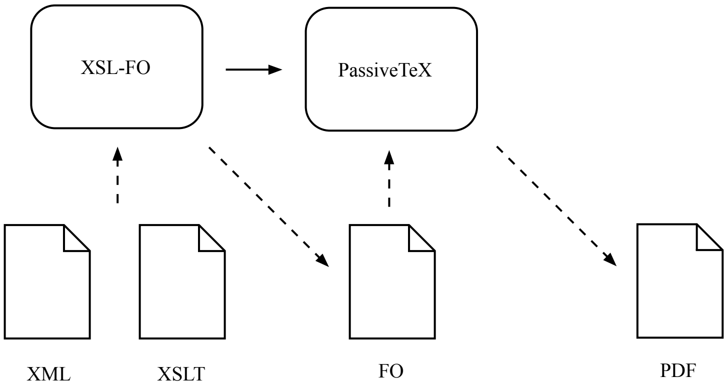
Within a TeX environment, xmltex can be a useful tool for XML processing. It offers a lightweight and declarative (apart from programmatic content manipulation where necessary, as mentioned above) syntax that TeX users should not be unfamiliar with. For XML users, not only the syntactical differences might prove to be a stumbling block .

4.2. PassiveTeX

Michel Goossens and Sebastian Rahtz were apparently not deterred by the challenges posed by xmltex. They've developed a library of TeX macros based on xmltex called PassiveTeX[7] that process XML documents that comply with the Formatting Objects (FO) schema[8].

Formatting Objects provides an XML vocabulary for formatting XML documents[9]. Unlike HTML, it has only presentational markup and compared to CSS, it contains also the complete content such as text, tables and images. The FO document is typically generated by XSLT and then passed to a FO formatter. The formatter renders the output format, most commonly this is a PDF. In the case of PassiveTeX, the FO formatter is TeX.

Figure 2. Passive TeX transformation



Here is an excerpt of the fotex.xmt from PassiveTeX. In this code snippet, a basic TeX document structure is associated with the root element of FO. A similarly named package

[5] David Carlisle (2000) Xml2tex. Accessing TeX. Available at https://ftp.agdsn.de/pub/mirrors/latex/dante/macros/xmltex/base/manual.html#manualN1059 (Accessed May 30, 2023)
[6] David Carlisle (2012) xm2tex source code on GitHub. Available at https://github.com/davidcarlisle/dpctex/tree/main/xmltex (Accessed: May 30, 2023)
[7] Michel Goossens and Sebastian Rahtz (2000): PassiveTEX. Available at https://ctan.org/pkg/passivetex (Accessed: May 30, 2023)
[8] W3C (2006) Extensible Stylesheet Language (XSL) Version 1.1. Formatting Objects. Available at https://www.w3.org/TR/xsl/#fo-section (Accessed: May 30, 2023)
[9] I'm not going to explain the vocabulary of Formatting Objects in any detail. For an introduction, please refer to: J. David Eisenberg (2001) Using XSL Formatting Objects. Available at https://www.xml.com/pub/a/2001/01/17/xsl-fo/index.html (Accessed May 30, 2023)

fotex.sty is referenced which adds macros and configuration options to render the FO document.

```
\XMLelement{fo:root}
  {}
  {\documentclass{article}
   \usepackage{fotex}
   \begin{document}
   \pagestyle{empty}
   \FOSetHyphenation
   %\ignorewhitespace
   }
  {\end{document}}
```

Just as an aside, FO and TeX overlap in terms of functionality. With PassiveTeX, you could decide whether you want to use FO's <fo:page-number-citation> to create a table of contents or LaTeX's built-in \tableofcontents macro. This applies to other elements such as counters, labels and other listings, too.

Compared to a pure xmltex approach, PassiveTeX has some benefits: You don't have to care about configuring xmltex, you only need to create FO from your XML input e.g., with a programming language with proper XPath support. If you already have an FO-based workflow, you can consider TeX as alternative to your existing FO formatter. A TeX-based FO formatter should have clear advantages when it comes to mathematical equations.

PassiveTeX is an experimental approach and has never seen wide acceptance, neither in the XML nor in the TeX communities. We can speculate about the reasons for its marginal adoption, but it seems likely that PassiveTeX just inherited the problems of its xmltex and FO foundations. Some general disadvantages of xmltex are mentioned above. In the context of PassiveTeX, Goossens and Rahtz criticized that the biggest problem with using xmltex for PassiveTeX is that it is complicated to extend the package[10]. Like xmltex, FO remained a technology for a specific niche, considered complex to master and only adopted by users already acquainted with XML. In addition, many developers are reluctant to use technologies that are not part of their usual stack. In this sense, it may not very appealing to XML developers to use TeX as an FO formatter, while TeX developers might prefer xmltex over converting XML to FO.

4.3. Pandoc

Pandoc is a popular Haskell library for converting markup formats into one another[11]. Pandoc can be invoked via an easy command line client and supports a variety of input and output formats, among others also XML formats like DocBook and JATS. Pandoc supports also LaTeX as output format and uses the Haskell library texmath[12] for converting MathML formulas to TeX.

With these features, Pandoc can be also used to convert from XML to LaTeX. In contrast to xmltex and PassiveTeX, you don't have to be a programmer to use Pandoc. Basic knowledge of how to use the command line is sufficient. To test Pandoc, I prepared a small DocBook test document.

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;<br/>
&lt;article xmlns="<link xlink:href="http://docbook.org/ns/docbook">http://docbook.org/ns/docbook</link>"
  xmlns:xlink="<link xlink:href="http://www.w3.org/1999/xlink">http://www.w3.org/1999/xlink</link>"
```

[10] Michel Goossens and Sebastian Rahtz (2000): PassiveTEX: From XML to PDF in TUGboat, Volume 21 (2000), No. 3—Proceedings of the 2000 Annual Meeting, p.224. Available at http://tug.org/TUGboat/tb21-3/tb68goos.pdf (Accessed: May 30, 2023)

[11] Pandoc (2023) Pandoc. A universal document converter. Available at https://pandoc.org/ (Accessed: May 30, 2023)

[12] John MacFarlane, Matthew Pickering (2023) texmath. Available at https://hackage.haskell.org/package/texmath (Accessed: May 30, 2023)

```
  version="5.0"&gt;
  &lt;title/&gt;
  &lt;section&gt;
    &lt;title&gt;Area of a Triangle&lt;/title&gt;
    &lt;equation&gt;
      &lt;math xmlns="<link xlink:href="http://www.w3.org/1998/Math/MathML">http://www.w3.org/1998/Math/MathML</link>" display="block"&gt;
        &lt;mi&gt;A&lt;/mi&gt;
        &lt;mo&gt;=&lt;/mo&gt;
        &lt;mfrac&gt;
          &lt;msup&gt;
            &lt;mi fontstyle="italic"&gt;a&lt;/mi&gt;
            &lt;mn&gt;4&lt;/mn&gt;
          &lt;/msup&gt;
          &lt;mn&gt;4&lt;/mn&gt;
        &lt;/mfrac&gt;
        &lt;mo&gt;&amp;#x22c5;&lt;/mo&gt;
        &lt;mroot&gt;
          &lt;mn&gt;3&lt;/mn&gt;
          &lt;mrow/&gt;
        &lt;/mroot&gt;
      &lt;/math&gt;
    &lt;/equation&gt;
  &lt;/section&gt;
  &lt;bibliography&gt;
    &lt;biblioentry&gt;
      &lt;citetitle&gt;Geometry Workbook For Dummies&lt;/citetitle&gt;
      &lt;author&gt;
        &lt;personname&gt;&lt;firstname&gt;Mark&lt;/firstname&gt;&lt;surname&gt;Ryan&lt;/surname&gt;&lt;/personname&gt;
      &lt;/author&gt;
      &lt;biblioid role="isbn"&gt;978–0471799405&lt;/biblioid&gt;
      &lt;pubdate&gt;2006&lt;/pubdate&gt;
      &lt;publisher&gt;
        &lt;publishername&gt;For Dummies&lt;/publishername&gt;
      &lt;/publisher&gt;
    &lt;/biblioentry&gt;
  &lt;/bibliography&gt;
&lt;/article&gt;
```

The results that Pandoc delivered were not very promising. Pandoc failed to convert MathML to TeX, even though it worked in other attempts. Moreover, Pandoc was not able to convert the bibliographical entry. While the MathML was quietly removed from the output, the contents of the bibliography were just printed as plain text but with an empty \section{} above. Pandoc's output is shown below:

```
\section{Area of a Triangle}
\section{}
Geometry Workbook For Dummies MarkRyan 978–0471799405 2006 For Dummies
```

On the Pandoc website, the tool is described as a Swiss army knife for markup formats. What applies to Swiss Army Knives applies to Pandoc, too: If the built-in tools are not suitable for the purpose, you have to get another tool yourself.

Pandoc provides an interface for users to write their own transformations, called filters. These are small programs that convert to or from Pandoc's intermediate abstract syntax tree (AST). Traditional Pandoc filters work on a JSON representation of the Pandoc AST and can be written in any programming language [13] . But the whole thing is very cumbersome: JSON has to be written to stdout and read from stdin and the filter only works if the programming language is also available on the user's system. Starting with version 2.0, Pandoc allows to write filters entirely in Lua which requires no external software to be installed.

Apart from the less declarative and time-consuming method of programming filters, Pandoc offers not very much extensibility. Unfortunately, there are also just a few configuration options to customize the output. Another downside is that you don't always get an error message when Pandoc is unable to process content. That leads me to the conclusion that Pandoc may not be suitable for professional scenarios, but might be useful for occasional conversions of lightweight documents where the users can fix the output themselves if necessary.

4.4. XSLT

XSLT[14] is well-known for its capabilities to convert from one XML vocabulary into another. Nevertheless, XSLT allows to output text-based content, too. Starting with XSLT 2.0

[13] Pandoc (2023) Pandoc Lua Filters. Available at https://pandoc.org/lua-filters.html (Accessed: May 30, 2023)

that became a W3C standard in 2007, its powerful query language XPath, grouping mechanisms and regular expressions make XSLT an interesting alternative for converting XML to LaTeX.

To output LaTeX in XSLT is quite simple. You need to add <xsl:output method="text"/> to your stylesheet. Then you can start writing your accustomed templates and associate XML nodes with TeX output. Here is a minimal example to demonstrate this method.

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;
&lt;xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0"&gt;
            &lt;xsl:output method="text"/&gt;
            &lt;xsl:template match="/article"&gt;
            &lt;xsl:text&gt;\documentclass{article}&amp;#xa;&lt;/xsl:text&gt;
            &lt;xsl:text&gt;\begin{document}&amp;#xa;&lt;/xsl:text&gt;
            &lt;xsl:apply-templates/&gt;
            &lt;xsl:text&gt;\end{document}&lt;/xsl:text&gt;
            &lt;/xsl:template&gt;
            &lt;/xsl:stylesheet&gt;
```

However, to program a full-fledged XML to LaTeX transformation will not remain as simple as the template above suggests. Templates to convert figures, tables and other elements will add up. Programming a transformation of MathML to TeX formulas will considerably be one of the biggest tasks. For anything in your XML documents that cannot be covered by standard LaTeX, you need to implement additional TeX packages that provide the required functionality. Depending on the Unicode support of your TeX engine, you also need to map Unicode characters such as ω (Greek Small Letter Omega U+03C9) to the corresponding TeX command *\omega*.

Given the extensibility of XSLT via imports and includes, it's easier to reuse the code and override it for other projects. But that's only true when the source XML does not change. For example, when you change from CALS to HTML tables, you basically find yourself completely rewriting your code. In any case, adapting your stylesheet to handle multiple XML vocabularies would impose a challenge.

5. An Alternative Approach

The methods presented in the previous section are more or less suitable for converting XML to TeX. They vary between ready-to-use applications that are difficult to configure (PassiveTeX, Pandoc) and programming approaches from scratch (xmltex, XSLT). The methods have all in common that they are not well-suited to be configured for arbitrary XML vocabularies.

When I was in the situation of knowing virtually nothing about TeX and given the task of developing an alternative to Chikrii's Word2TeX plugin[15] for Microsoft Word eight years ago, I've also been thinking of a way to handle arbitrary XML vocabularies by one library. But I learned from my colleagues in the typesetting department that the library must be configurable not only for different ML inputs, but for different TeX outputs as well: Depending on the customer or product, other TeX packages and engines are used. For example, we use LuaTeX with unciode-math[16] for the conversion of NISO STS standards to TeX while we still use pdflatex for traditional typesetting projects in TeX.

[14] W3C (2017) XSL Transformations (XSLT) Version 3.0. Available at https://www.w3.org/TR/xslt-30/ (Accessed: May 30, 2023)
[15] Chikrii (2022) Word2Tex. Available at http://www.chikrii.com/products/word2tex/ (Accessed: May 30, 2023)
[16] Will Robertson (2020) Experimental Unicode mathematical

typesetting: The unicode-math package. Available at: https://ctan.org/pkg/unicode-math (Accessed: May 30, 2023) xml2tex is a module of the le-tex transpect [17] framework and covers various aspects of converting arbitrary XML to TeX-based formats. It is based on XProc and XSLT and built around an XML vocabulary agnostic configuration that allows to associate XML contexts with TeX instructions. xml2tex consists of three more or less configurable components:

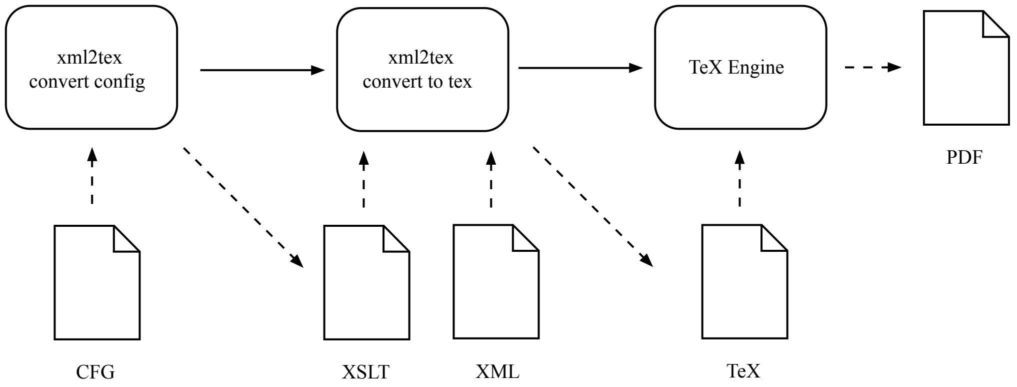
1. Convert CALS/HTML tables to tabular or htmltabs[18]

2. Transform MathML to TeX

3. Transform XML to TeX

The components for converting mathematical formulas and tables are prefabricated XSLT stylesheets that are only configurable to some extent. The third transformation is based on a configuration.

Figure 3. xml2tex conversion pipeline



Convert CALS/HTML tables to tabular or htmltabs

First, CALS tables are normalized. Therefore, we adopted Andrew J. Welch's table normalization[19] and implemented it for CALS tables. If the input consists of HTML tables, they are converted to CALS before. The table normalization facilitates converting the tables to TeX later.

```
+-----------+-----------+      +-----+-----+-----+-----+
| a         | b         |      | a   | a   | b   | b   |
|           +-----+-----+      +-----+-----+-----+-----+
|           | c   | d   |      | a   | a   | c   | d   |
+-----------+-----+     |  =>  +-----+-----+-----+-----+
| e               |     |      | e   | e   | e   | d   |
+-----+-----+-----+     |      +-----+-----+-----+-----+
| f   | g   | h   |     |      | f   | g   | h   | d   |
+-----+-----+-----+-----+      +-----+-----+-----+-----+
```

Depending on the used options, another stylesheet converts the normalized tables either to tabular or htmltabs tables. The TeX tables are inserted as processing instructions into the original XML document.

[17] le-tex (2023) transpect framework documentation. Available at https://transpect.io (Accessed: May 30, 2023)

[18] transpect (2023) htmltabs source code. Available at https://github.com/transpect/xerif/blob/main/latex-oops/htmltabs.sty (Accessed: May 30, 2023)

[19] Andrew J.Welch (2006) Table Normalization in XSLT 2.0. Available at http://ajwelch.blogspot.com/2006/09/table-normalization-in-xslt-20.html (Accessed: May 30, 2023)

5.1. Math Transform MathML to TeX

Second, MathML equations are normalized and transformed to TeX. An XProc library entitled mml-normalize has a number of built-in heuristics to normalize badly authored equations e.g., resolving superscripts with an empty base symbol. After this step, the XProc library mml2tex transforms MathML and inserts the TeX equations as processing instructions into the XML document. When you take the example from the Pandoc section as input, this would be the output of mml2tex:

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;
&lt;article xmlns="http://docbook.org/ns/docbook" version="5.0"&gt;
  &lt;title&gt;Area enclosed by a circle&lt;/title&gt;
  &lt;equation&gt;
    &lt;?mml2tex A=\frac{a^{4}}{4}\cdot \sqrt{3}?&gt;
  &lt;/equation&gt;
&lt;/article&gt;
```

The third step is the central aspect of an xml2tex conversion. The xml2tex configuration is converted to an XSLT stylesheet which is applied on the XML input document later. But let me walk you through the main aspects of our xml2tex configuration.

5.2. Transform XML to TeX

The xml2tex configuration vocabulary is specified by an RelaxNG schema[20]. Here is a minimal example configured for DocBook:

```
&lt;?xml version="1.0" encoding="UTF-8"?&gt;<br/>
  &lt;set xmlns="<link xlink:href="http://transpect.io/xml2tex">http://transpect.io/xml2tex</link>" xmlns:xsl="<link xlink:href="http://www.w3.org/1999/XSL/Transform">http://www.w3.org/1999/XSL/Transform</link>"&gt;
  &lt;import href="../docx2tex/conf/conf.xml"/&gt;
  &lt;ns prefix="dbk" uri="<link xlink:href="http://docbook.org/ns/docbook">http://docbook.org/ns/docbook</link>"/&gt;
  &lt;preamble&gt;
    \documentclass{article}
    \usepackage[utf8]{inputenc}
    \usepackage[main=english,greek]{babel}
  &lt;/preamble&gt;
  &lt;front&gt;
    \title{&lt;xsl:value-of select="/dbk:article/dbk:info/dbk:title"/&gt;}
    \maketitle
  &lt;/front&gt;
  &lt;back&gt;
    \printindex
  &lt;/back&gt;
  &lt;template context="dbk:chapter"&gt;
    &lt;rule name="chapter" type="env"&gt;
      &lt;param select="dbk:title"/&gt;
      &lt;text select="* except dbk:title"/&gt;
    &lt;/rule&gt;
  &lt;/template&gt;
  &lt;regex regex="([&amp;#x370;-&amp;#x3ff;]{2,})+"&gt;
    &lt;rule name="foreignlanguage" type="cmd"&gt;
      &lt;param&gt;greek&lt;/param&gt;
      &lt;param regex-group="1"/&gt;
    &lt;/rule&gt;
  &lt;/regex&gt;
  &lt;charmap&gt;
    &lt;char character="&amp;#x3c9;" string="${\omega}$"
          context="*[@css:font-style eq 'italic']"/&gt;
  &lt;/charmap&gt;
&lt;/set&gt;
```

The xml2tex configuration contains eight top-level elements. The first five specify the basic document structure:

- < ns/> works similar as it’s Schematron pendant and defines namespaces.
- <import/> can import other xml2tex configurations.
- <preamble/> is specifies TeX document class, packages and configuration options.
- The contents of < front/> are placed between \begin{document} and before the actual content.
- <back/> is a placeholder for anything that comes after the text body and before \begin{document}.

The actual mapping between XML nodes and text is performed by these three templates:

[20] Transpect (2023) xml2tex RelaxNG schema. Available at: https://github.com/transpect/xml2tex/blob/master/schema/xml2tex.rng (Accessed: 30 May, 2023)

- <template/> associate XML nodes with TeX instructions.
- <regex/> strings matched by a regular expression are replaced with a TeX instruction.
- <charmap/> replaces Unicode characters in certain XML contexts with TeX instructions (or other strings).

xml2tex cannot deny its XSLT origins, so let us take a closer look at the anatomy of the chapter template. Internally, it will be converted to an XSLT template matching on “dbk:chapter”. But in contrast to XSLT templates, its contents are more restricted. Based on is @type attribute, the <rule/> element inserts either a TeX command (\chapter) or environment (\begin{chapter}…\end{chapter}). Then you can specify whether the TeX instruction is followed by a number of arguments (<param/>). options (<option/>) or regular text (<text/>). You can insert static text or specify an XPath with the @select attribute.

```
&lt;template context="dbk:chapter"&gt;
  &lt;rule name="chapter" type="env"&gt;
    &lt;param select="dbk:title"/&gt;
    &lt;text select="* except dbk:title"/&gt;
  &lt;/rule&gt;
&lt;/template&gt;
```

With this mechanism, you can take arbitrary XML and configure your preferred TeX output. Compared to pure XSLT, this is a more declarative approach which resembles the structure of a TeX document. In contrast to xmltex, you have more flexible configuration options and XPath as a powerful query language. Furthermore, it’s also possible to insert XSLT code within the xml2tex configuration, for example if you want to encapsulate the evaluation of the TeX list type.

xml2tex has proven itself at le-tex in productive use for different customers with different XML vocabularies and TeX requirements. Even though le-tex is also using xmltex for some workflows, our TeX developers are more convinced of the xml2tex approach, in particular because it relieves them of configuring xmltex.

5.3. Issues

None, of course (except the ones listed below):

- To prevent the user from fixing ambiguous template matches (they are not presented to the user by the XProc processor), the templates automatically got a priority assigned, meaning templates placed below are more important than those placed above. This can be cumbersome and I think I’ll drop this with the switch to XProc 3.0.
- When other configurations are imported, they are merged together into one XSLT stylesheet and the templates of the importing configuration are assigned with a higher priority than the imported ones. This makes it harder to selectively override imported templates. But it’s currently not possible with XProc to reference within a dynamically generated XSLT other generated XSLTs as imports (except when written to disk, which is too “antsy” for an XProc guy).

6. Conclusion

This paper presented the challenges of converting from XML to TeX. Even if the general structure of XML and TeX seem simple at first glance, math, tables, different TeX engines and packages make configuring a converter very complex. Speedata, ConTeXt[21] and other approaches could also have been presented, but that would no longer have been within the scope of this paper.

There exist various methods for the task, ranging from lightweight converters to powerful programming languages. However, many of the solutions are experimental, immature and/or require additional programming work. Besides, the versatility of XML on the one hand and TeX on the other hand makes it difficult to find a universal approach.

The presented xml2tex[22] library addresses the problem by providing ready-to-use conversions for specific problems and, on the other hand, introducing an XML-based, declarative configuration vocabulary that can be used to create a common interface between XML and TeX.

[21] Hans Hagen (2001) context – The ConTeXt macro package. Available at https://ctan.org/pkg/context (Accessed: May 30, 2023)

[22] A TeX version of this paper converted with xml2tex can be downloaded here: https://xporc.net/wp-content/uploads/2023/05/MarkupUK/Martin_Kraetke_Bridging-the-Gaps-Between-TeX-and-XML.tex

Markup UK

Building a cloud-based visual operating system entirely based on XML

Leveraging the capabillities of the XIOS/3 and CloudBackend platforms

Daniel Arthursson, CloudBackend.com

Martin Nilsson, xios3.com

Interoperability and extensibility are keywords for XML. Why are we not seeing the same for software applications, web applications, mobile apps, and operating systems in general? Why can't I run an iPhone app om my Samsung TV? Why isn't Mac applications possible to run on my Windows computer? If we had interoperability and extensibility in a similar way that XML provides for data for software, we would not have any of these problems. Software would run across all types of devices, screen form factors, and operating systems – across desktop, mobile, smart TVs, and the infotainment systems of cars.

This paper discusses how to use the XIOS/3 Edge Application Platform and the CloudBackend Singularity Database to create a new XML- and cloud-based operating system complete with productivity applications, software development tools, and a file system – including extensive support for XML. While CloudTop XML OS is still under development, this paper provides a snapshot of the current state of the implementation. It challenges the perception of what XML can be used for.

1. Introduction

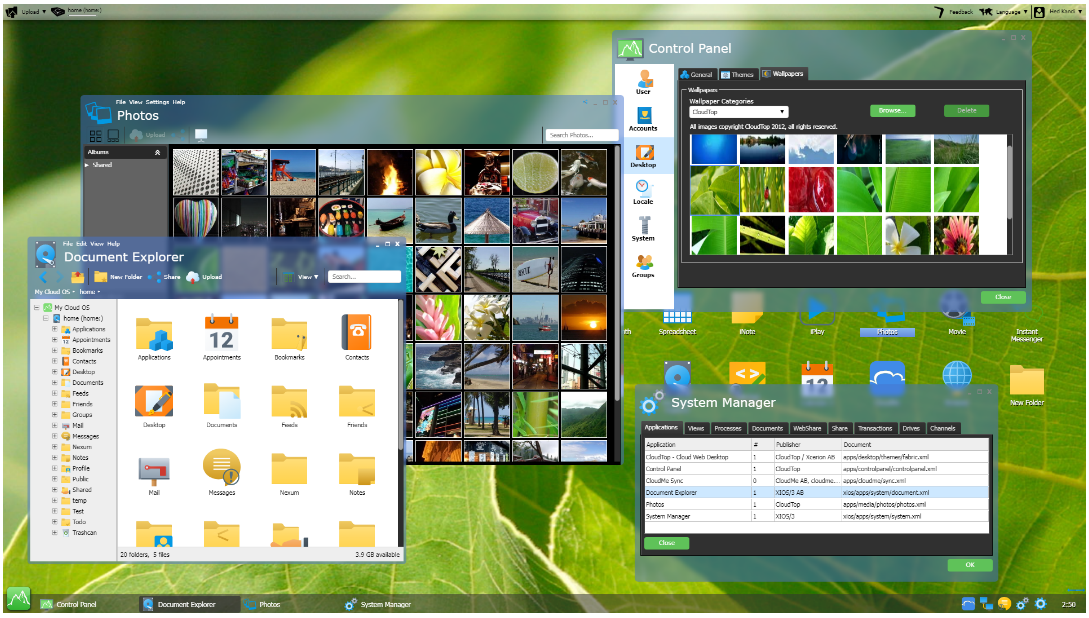
Many years ago, we set out with the goal to simplify development, making applications work across devices, and use XML to build applications instead of traditional programming. With a background in building an XML Application Server and an XML-based web server we had seen the promise of XML, and how the hierarchical structure of XML itself helped to solve many recurring problems. The time had come to apply XML to software that was supposed to run on Windows, Mac, Linux, mobile devices, and other types of devices.

We wanted to break out of the boundaries enforced by native compiled code for a specific processor architecture and the underlying operating system. We wanted applications that would run equally well on Windows, as on Mac or Linux. We wanted applications that could change their form factor to also run on mobile devices. We wanted the applications of the operating system to store all its data in tagged XML markup that nourished interoperability between applications and long-term storage and retrieval of information while simplifying search.

We concluded that it would not be good enough to create a cross-device, cross-operating system development framework, we needed new data storage technology to replace the outdated xomputer file systems with an XML repository, a new set of productivity applications for office workers that breathed XML, and we needed a collaborative XML-based operating system to tie it all together to complete our vision.

This is a long-term vision and we are not at the end of it yet, but we are step by step humbly getting closer to realizing it. We got the XML development platform with XIOS/3, we got the XML repository and Singularity Database with CloudBackend. What we are missing is the completion of the productivity applications and the cloud-based XML operating system, CloudTop.

Figure 1. CloudTop cloud-based desktop and applications built entirely in XML.



Wallpapers
Wallpaper Categories: CloudTop
All images copyright CloudTop 2012, all rights reserved.

System Manager

Applications | Views | Processes | Documents | WebShare | Share | Transactions | Drives | Channels

Application	#	Publisher	Document
CloudTop - Cloud Web Desktop	1	CloudTop / Xcerion AB	apps/desktop/themes/fabric.xml
Control Panel	1	CloudTop	apps/controlpanel/controlpanel.xml
CloudMe Sync	0	CloudMe AB, cloudme....	apps/cloudme/sync.xml
Document Explorer	1	XIOS/3 AB	xios/apps/system/document.xml
Photos	1	CloudTop	apps/media/photos/photos.xml
System Manager	1	XIOS/3	xios/apps/system/system.xml

20 folders, 5 files
3.9 GB available

CloudTop is the answer to the question, how would you design a new operating system if you knew the Internet and XML existed? The OS would be delivered over the Internet and always be up-to-date without system updates. Any applications written in it would allow for collaboration with other people over the Internet. All information stored in the OS would be in XML format and any data exchanged for collaboration would be in XML. Applications would be able to open each other's data and transform it into its required format, possibly allowing a word processing application and a presentation application to only be two different views into the same XML document.

This paper will introduce a few applications built for CloudTop and some of the challenges with using XIOS/3 and CloudBackend to build CloudTop.

2. Finding a cure to the chaotic software landscape

Curing today's mess with incompatibility across operating systems and devices will require quite some dramatic changes. It will require software to be rewritten from the ground up using XML instead, data storage formats to be migrated to XML, and the willingness to embrace openness, as with XML your client source code will no longer pretend be a secret. Anyone may open the source code of your application and learn how to make their own, like how HTML works. If the server allows it they may also use XLink to link into your application and create a composite application or mashup using parts of your application.

The cure is nevertheless needed. Software-as-a-Service (SaaS) companies, enterprises, and mobile app developers spend far too much money on building web applications, iPhone apps, Android apps, and even possibly locally installed desktop applications. Rebuilding the same application for a multitude of form factors and operating systems is expensive and time-consuming. Using XML only a single truth would be needed, a single source code that could feed all devices. This would reduce time to market through reduced development time, but it would save an equal amount on maintenance costs. Making the trade-off to make the code open in XML, might enable companies to build applications that they otherwise would not receive a budget for. Since many companies already have migrated their software to web versions built in JavaScript, which have an open visible source code, the openness part of the equation might prove to be less difficult than we initially imagined.

The last part is that for a new operating system to gain traction, it needs to have enough software with expected functionality to make a shift possible for an end-user. Gaining a critical mass of available software and making it easy to build software for the OS is therefore a crucial component. Our ace up our sleeve is the thousands of already available XML applications out there for almost any imaginable use case. If we can make them easy to register in the underlying XML repository of CloudBackend and then allow a user interface to quickly be built on top using XIOS/3, we would find a shortcut to get mission-critical software available on the operating system.

Effectively CloudTop makes its users create XML without seeing it. Any application running on the platform is thus like a custom XML editor, tailored for its specific XML application. The authored XML is stored in something that feels like a file system for the user, but in reality is the CloudBackend Singularity Database, which in the context of CloudTop acts as an XML repository.

3. What is CloudTop really?

CloudTop is a new cloud-based operating system and virtual desktop surface written in XML on top of XIOS/3 Edge Application Platform and the CloudBackend Singularity Database. It is delivered through any traditional web server without any server dependencies. It then runs within the web browser where the entire desktop surface is created from XML and additional applications are running as separate windows on top of that, all within a single web browser tab.

The type of virtual desktop provided by CloudTop is not a traditional operating system virtualization technology like Citrix or Microsoft Terminal Server. Traditional virtual desktop technologies run the host operating systems in the cloud on a server and then screencast the desktop and applications into a web browser or dedicated native client. With CloudTop the OS and its applications are all executing and running within the web browser thanks to the device edge computing capabilties of XIOS/3. The cloud and server are, thanks to CloudBackend, transformed into an authentication engine to load, save, and coordinate delta changes of XML documents for collaboration purposes.

Applications built for CloudTop can run entirely within the browser if built in XML and only using XML documents as its storage format. They can also interact with APIs in the cloud like XML Web Services (SOAP) or REST APIs (JSON or XML). Having portions of the application logic in the cloud behind an API will of course make the application more sensitive to disruptions and introduce latency for waiting on server responses. However, all user interactions can be quickly resolved and managed on the client, since the remaining application logic is written in XML and executed within the web browser on the device.

CloudTop has things you normally associate with an operating system such as a file system, processes, applications, security, users, command line interface, caches, user interface rendering, SDK for software developers, and network communication. It does not however include the lower levels of an OS like the kernel, BIOS, and things needed to make it boot a computer. Thus CloudTop can be said to include the higher levels of a traditional OS. Our reasoning here is that the most effect for end-users and developers is achieved at the higher levels, software SDK and development tools, and communication with the outside world. The lower levels can be reused from Linux, Android, Windows, or any operating system. CloudTop only needs something that boots and opens a web browser like the open source WebKit browser and then starts CloudTop in fullscreen mode. One implementation of such an environment is the Google Chrome OS. The added benefit of this is that anyone with a web browser may run CloudTop without installing anything. VPN and security protocols can also be added to the lower-level boot OS, though relevant features are constantly added to the browser environment.

4. Changing the perception of a computer

The applications, the run-time state, the data, and the delta changes to data are all XML. This allows the run-time state of applications to be synchronized across devices and across different logged-in identities. The effect of this is that applications almost become like a virtual machine or container that can move between two physical servers while running, i.e. application virtualization makes moving applications between devices possible thanks to XML.

If you can move a running application window from one laptop to another, or to your phone, your computer is essentially no longer running on a single hardware. It is in the cloud and can manifest itself on any device and treat it like a piece of glass. Multiple devices can together form a multi-screen setup and parts of an application can run on different devices for a more user friendly-experience.

If applications can float freely between all devices and they together make up your total computing experience, then what is a computer? The XML-based cloud operating system makes this possible without any effort for the software developer to build an application. We see this as the true promise of cloud computing and ubiquitous computing, using the device's local processor, but treating all devices as screens to your cloud computer - pieces of glass.

5. Verifying our assumptions

While building on the CloudTop, we needed to verify that the cloud OS really could be used create everything from a presentation application to an Integrated Development Environment (IDE), to applications like mail, calendar, instant messaging, and photo viewer. Was really all the UI components needed to build such complex applications available in XIOS/3 XML languages for developing? Could the CloudBackend Singularity Database really serve the content needed for emails, calendar, photos, and word processing?

Along the way we have built some 30 applications on the platform and feel confident that the concept of applications being developed in XML and executing in XML works. Both from a development perspective, it is convenient to develop these applications and possible to maintain them, but more importantly, that it provides fast enough applications. It would be pointless if we had all the goodness of XML, but the applications became unbearably slow.

6. Looking through a few of the sample applications built

There would not be enough space in an article to go through all applications that have been built for CloudTop, but we will go through a few of them as each one of them introduces interesting and important concepts for the CloudTop OS and how it makes use of XML.

- XMLPad - Data Manipulation and Transactions
- Kanban - Hierarchical Data Model
- Contacts - Key/values, Meta-data, and Datatypes

◇ CloudTop - Combining Applications into a Desktop

The above list provides a brief introduction. Additional information is available through the CloudBackend service and the SDK Web UI (XIOS/3) and CloudBackend developer documentation. Using CloudBackend you can write and run applications on your own.

CloudTop is a separate project and not part of either the CloudBackend and XIOS/3 companies. They aim to release a new version of the CloudTop desktop using the latest XIOS/3 and CloudBackend by the end of this year. Any application developed as a single-page application on the CloudBackend Singularity Database today can run within the CloudTop desktop when it is released.

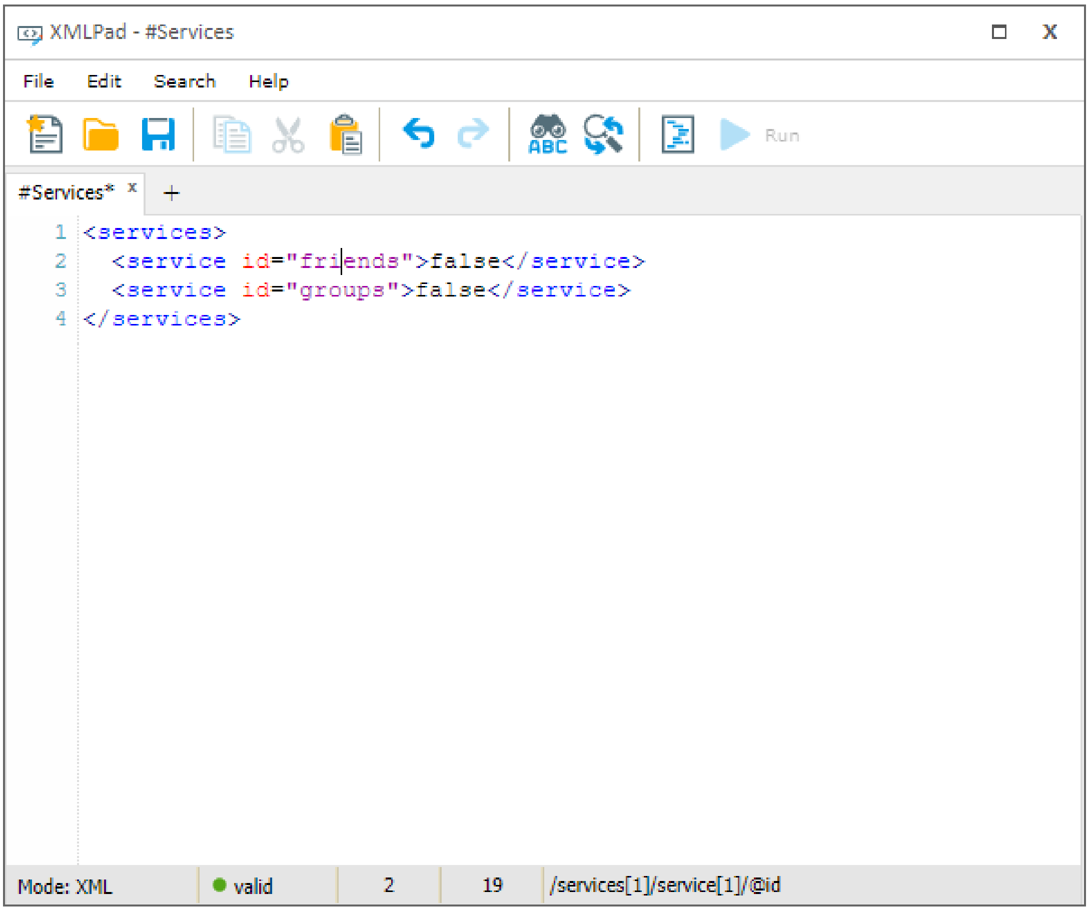
7. XMLPad - Data Manipulation and Transactions

One of the most important productivity applications on any system is the ability to write text in various forms, from the humble notepad to WYSIWYG (what you see is what you get) word processors, and code and markup editors with support for syntactic and semantic validation. Several such applications were made using XIOS/3 with various levels of sophistication, but let us look at “XMLPad”, a simple XML editor. At its core it is a presentation and input component that is bound to a document in memory. Every change in the document is automatically reflected on the screen for the user, but any input the user makes goes through the XIOS/3 transaction engine. These changes are expressed as a series of modifications at points in the document pointed out by XPath expressions. This allows anyone else interested in this document to get notified with only the delta of what has changed. If the document is shared with another user or device only these changes need to be transmitted.

There are many hidden details and edge cases on regulating change frequencies and coalescing change sets, but one of the more interesting for general use is how to handle syntactically invalid XML markup that inevitably happens when a user is typing. To solve that the offending section of the document can be wrapped in a special “not valid” namespace and encoded with entities or wrapped in a CDATA block, and can then be handled as any other XPath-indicated XML change set.

Besides the advanced applications like having servers or other clients apply these transactions elsewhere for collaborative applications or auditing change logs, they also can serve an immediate use for the application developer. A transaction log on a document can serve as a undo buffer if the transactions are applied in reverse. If then those transactions are moved to a different log they serve as a redo buffer.

Figure 2. The XMLPad application.



```
<services>
  <service id="friends">false</service>
  <service id="groups">false</service>
</services>
```

8. Kanban - Hierarchical Data Model

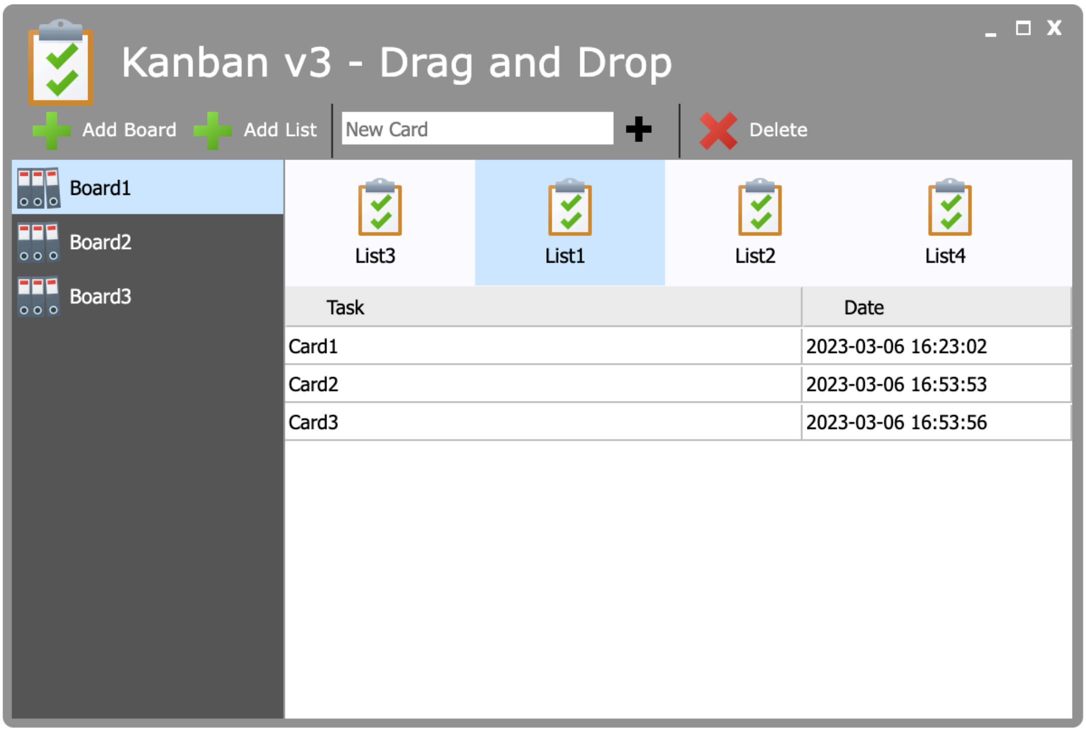
XML documents are already structured data of course, but putting XML documents into hierarchical structures has proven to be very beneficial for many applications. While XML documents are great for collecting individual related data points and processing them as a unit, there are some real-world considerations. The main one is in the security model where it is impractical to make only some parts of a document readable or writeable while still maintaining working XPath expressions, XSLT, etc. Besides access meta-data, other meta-data like timestamps are better managed outside of the document itself. While it isn't always clear-cut where the line should go between XML elements and individual documents, there is also great value in arranging the data in a conceptually different hierarchy despite being able to in practice put everything in a giant XML document.

This illustrates the need to have a data model outside of the XML documents, that brings additional meta-data, security, and possible sharing of XML documents into the picture. Being able to put several XML documents into a container, having sub-containers to classify different XML documents as something else and being able to move documents across containers to denote different meanings to them are all features desirable.

In our development tutorial we have our version of creating a Todo app, a drag-and-drop Kanban board application. For this application we decided to break out the data model into Board containers, and List containers in the CloudBackend database. Each Board represents one kanban view, each List represents the different swim lines on the Board. The XML files stored in the List containers represent the individual tasks. This makes it easy to move a task (card) between two lists, it is a matter of only moving the document to a new container.

Since XIOS/3 is built on the premise of using XML for all data interactions the containers residing in the CloudBackend database are exposed to the developer as XML atom feeds with container elements containing links to other container nodes in a tree representing the data mode of the database. This allows the "Board" list to bind to the node set of Board container nodes, the "List" list to the node set of the container for the selected Board, and finally the swim line to the respective Lists. Add to that a simple form dialog to create new tasks (cards) and a delete action and we have a rudimentary Kanban application in 100-200 lines of code.

Figure 3. The tutorial Kanbon application.



Task	Date
Card1	2023-03-06 16:23:02
Card2	2023-03-06 16:53:53
Card3	2023-03-06 16:53:56

The following is the entire code for the applicatio above.

```
<application name="myKanban3" icon="icon://rocket" instances="0" theme="fabric">
  <view name="Kanban" title="Kanban v3 - Drag and Drop" width="600" height="400" icon="icon://clipboard_checks" winstate="false">
    <style>
      #boards.iconlist .layout_submenu .text { color: #FFF; }
      #boards.iconlist .marble .listselected .text { color: #000; }
      #lists.iconlist .layout_sectionlist .text { margin-top: 4px; font-weight: normal; }
    </style>
    <toolbar name="kanbanBar">
      <group name="leftBar">
        <buttonbox name="addBoard" text="Add Board" icon="icon://plus"/>
        <buttonbox name="addList" text="Add List" icon="icon://plus"/>
      </group>
      <group name="cardBar">
        <input name="addCard" placeholder="New Card" icon="icon://plus"/>
      </group>
      <group name="rightBar" align="right">
        <buttonbox name="deleteCard" text="Delete" icon="icon://delete"/>
      </group>
    </toolbar>
    <panel name="MainPanel" type="column">
      <iconlist name="boards" width="150" height="100%" layout="submenu" deselect="false" scroll="false" iconsize="24" style="background-color: #555;">
        <rule match="fs:folder">
          <item text="{@name}" icon="icon://folders2?color=fff"/>
        </rule>
      </iconlist>
      <panel type="row" width="100%" bgcolor="#fafaff">
        <iconlist name="lists" width="100%" height="70" layout="sectionlist" scroll="true" deselect="false" iconsize="32">
          <rule match="fs:folder">
            <item text="{substring-after(@name, ' ')}" icon="icon://clipboard_checks"/>
          </rule>
        </iconlist>
        <panel name="todoPanel" type="flow" height="100%" width="100%" bgcolor="#ccc">
          <grid name="cards" height="100%" draggable="true">
            <row match="atom:entry">
              <column name="subjectCol" match="atom:title" display="substring-before(.,'.xml')" label="Task" filter="true" width="100%"/>
              <column name="dateCol" match="atom:updated" display="translate(., 'TZ', '  ')" label="Date" filter="true" width="140"/>
            </row>
          </grid>
        </panel>
      </panel>
    </panel>
  </view>
```

```xml
<process name="Kanban - Process">
  <trigger view="Kanban" event="Loaded" step="init"/>
  <trigger view="Kanban" component="boards" event="Select" step="selectBoard"/>
  <trigger view="Kanban" component="lists" event="Select" step="selectList"/>
  <trigger view="Kanban" component="addCard" event="Select" step="addCard"/>
  <trigger view="Kanban" component="addCard" event="Enter" step="addCard"/>
  <trigger view="Kanban" component="addBoard" event="Select" step="addBoard"/>
  <trigger view="Kanban" component="addList" event="Select" step="addList"/>
  <trigger view="Kanban" component="deleteCard" event="Select" step="deleteCard"/>
  <trigger view="Kanban" component="lists" event="Drop" step="dropList"/>
  <trigger view="Kanban" component="boards" event="Drop" step="dropBoard"/>

  <step id="init">
    <operation name="bind" value="tenant://Kanban">
      <component view="Kanban" name="boards" select="/atom:feed"/>
    </operation>
    <operation name="setSelection" value="">
      <component view="Kanban" name="boards">
        <item select="/atom:feed/fs:folder[1]"/>
      </component>
    </operation>
  </step>
  <step id="selectBoard">
    <operation name="bind" value="{#Kanban#boards#@path}">
      <component view="Kanban" name="lists" select=""/>
    </operation>
    <operation name="setSelection" value="">
      <component view="Kanban" name="lists">
        <item select="/atom:feed/fs:folder[1]"/>
      </component>
    </operation>
  </step>
  <step id="selectList">
    <alias name="testAlias" value="#Kanban#boards"/>
    <operation name="bind" value="{#Kanban#lists#@path}">
      <component view="Kanban" name="cards" select=""/>
    </operation>
  </step>
  <step id="addBoard">
    <operation name="confirm">
      <type value="input"/>
      <message>Please enter the name of the new Kanban board.</message>
      <modal>false</modal>
      <title>Enter board name</title>
      <ok step="createBoard" text="OK"/>
      <cancel step="cancel" text="Cancel"/>
    </operation>
  </step>
  <step id="addList">
    <operation name="confirm">
      <type value="input"/>
      <message>Please enter the name of the new board list.</message>
      <modal>false</modal>
      <title>Enter list name</title>
      <ok step="createList" text="OK"/>
      <cancel step="cancel" text="Cancel"/>
    </operation>
  </step>
  <step id="createBoard">
    <alias name="containerName" value="{!}"/>
    <operation name="filesystem" value="tenant://Kanban/">
      <create type="folder" name="{$containerName}"/>
    </operation>
  </step>
  <step id="createList">
    <alias name="containerName" value="{!}"/>
    <operation name="filesystem" value="{#Kanban#boards#@path}">
      <create type="folder" name="{$containerName}"/>
    </operation>
  </step>
  <step id="addCard">
    <alias name="cardName" value="{!}"/>
    <operation name="filesystem" value="{#Kanban#lists#@path}">
      <create type="file" name="{$cardName}.xml" rename="false">
          <content type="text/xml">
              <card deadline="" completed="false">{$cardName}</card>
          </content>
      </create>
    </operation>
  </step>
  <step id="deleteCard">
    <alias name="url" model="string" value="{#Kanban#cards#atom:content/@src}"/>
    <operation name="decision" value="$url">
      <when test="'{$url}' != ''">
        <operation name="filesystem" value="$url">
          <delete permanent="true"/>
        </operation>
      </when>
      <otherwise>
        <operation name="confirm">
          <type value="message"/>
          <message>Please select a card first to delete.</message>
          <modal>false</modal>
          <title>No card selected</title>
          <icon>icon://delete</icon>
        </operation>
      </otherwise>
    </operation>
  </step>
  <step id="dropList">
    <alias name="dropSource" value="!"/>
    <alias name="dropSourceDoc" value="!" model="value"/>
    <alias name="dropTargetFolder" value="{#Kanban#lists#@path}"/>
    <operation name="decision" value="$dropSource">
      <!-- Check that we are not dropping a board or list, i.e. a container -->
      <when test="local-name() != 'folder'" step="handleDrop"/>
    </operation>
  </step>
  <step id="dropBoard">
    <alias name="dropSource" value="!"/>
    <alias name="dropSourceDoc" value="!" model="value"/>
    <alias name="dropTargetFolder" value="{#Kanban#boards#@path}"/>
    <operation name="decision" value="#Kanban#boards">
      <!-- Check that we are dropping a list and not a board into another board, only accept lists -->
      <when test="not(../fs:folder[@id = '{$dropSource#@id}']) and '{local-name($dropSource)}' = 'folder'" step="handleDrop"/>
    </operation>
  </step>
  <step id="handleDrop">
    <operation name="decision">
```

```
        <!-- Check that what we are dragging is not the same as where we are dropping -->
        <when test="'{$dropSourceDoc}' != '{$dropTargetFolder}' and '{$dropSource#@path}' != '{$dropTargetFolder}'">
          <operation name="filesystem" value="$dropSource">
            <move to="{$dropTargetFolder}" silent="false" progress="true"/>
          </operation>
        </when>
      </operation>
    </step>
  </process>
</application>
```

9. Contacts - Key/values, Meta-data, and Datatypes

The CloudBackend database design pattern of having data objects in a searchable, hierarchical containers, which then contain objects that can have zero to many streams attached like an XML document, JSON, text, or binary, allows for a flexible data model.

Examples of what containers can represent could be; invoices, calendar, contacts, time reporting, conference room booking, recipe database, truck journal, and so on. Each stored object would then have an XML document and an XML schema that represent its structure.

To enable rapid searches, datatypes can be defined that automatically extract information using XPath expressions from XML documents as soon as they are stored in the meta-data of the object, which is stored as key/value pairs in the data model. This allows both searching and the listing of a container to result in an ATOM feed that contains all data needed to present the list using XIOS/3 built-in UI components or XSLT.

The ATOM feed structure can be bound directly to the UI components or to a component that applies an XSL transformation to present the content as HTML. Any changes to the data model of the ATOM feed will then automatically update the user interface, meaning that if a new XML document is added, deleted, or updated in the container, it will immediately be reflected in the user interface.

As an example, the contact manager application uses an XML version of the vCard format to store contacts in individual XML documents. The datatype declaration for these vCard files has the following meta-data exctraction (indexing) instructions.

```
<index>
    <dc:firstname xpath="/vcard/n/given"/>
    <dc:lastname xpath="/vcard/n/family"/>
    <dc:nickname xpath="/vcard/nickname"/>
    <dc:homephone xpath="/vcard/tel/home"/>
    <dc:mobilephone xpath="/vcard/tel/cell"/>
    <dc:homecountry xpath="/vcard/adr/home/ctry"/>
    <dc:homemail xpath="/vcard/email/home"/>
    <dc:homestreetaddress xpath="/vcard/adr/home/street"/>
    <dc:homezipcode xpath="/vcard/adr/home/pcode"/>
    <dc:homecity xpath="/vcard/adr/home/city"/>
    <dc:businesscompanyname xpath="/vcard/org/orgname"/>
    <dc:businessdepartment xpath="/vcard/org/orgunit"/>
    <dc:businessphone xpath="/vcard/tel/work"/>
    <dc:businessfax xpath="/vcard/tel/fax/work"/>
    <dc:businessjobtitle xpath="/vcard/title"/>
    <dc:businessmail xpath="/vcard/email/work"/>
    <dc:linkedin xpath="/vcard/linkedIn"/>
    <dc:facebook xpath="/vcard/facebook"/>
    <dc:twitter xpath="/vcard/twitter"/>
    <dc:card xpath="/vcard/@card"/>
    <dc:workmobile xpath="/vcard/tel/cellwork"/>
    <dc:businessindustry xpath="/vcard/org/orgindustry"/>
    <ni:photo xpath="/vcard/photo"/>
  </index>
```

Whenever a document is updated or uploaded these XPath expressions are applied to the data model in CloudBackend and the extracted properties are stored in the search index as key/value pairs. Just the ATOM feed itself is then enough to present a usable overview of the objects in the container collection.

```
<atom:feed xmlns:atom="http://www.w3.org/2005/Atom" xmlns:os="http://a9.com/-/spec/opensearch/1.1/" xmlns:dc="http://xcerion.com/directory.xsd" xmlns:ni="http://xcerion.com/noindex.xsd" dc:folder="562958574623716" xmlns:fs="http://xcerion.com/folders.xsd">
  <os:totalResults>1</os:totalResults>
  <os:startIndex>0</os:startIndex>
  <os:itemsPerPage>100</os:itemsPerPage>
  <atom:entry>
    <atom:title>Karl Hyltberg.xml</atom:title>
    <atom:published>2021-03-31T09:10:45Z</atom:published>
    <atom:updated>2023-05-29T11:06:39Z</atom:updated>
    <atom:link rel="alternate" type="text/xml" href="https://api.cloudbackend.com/v1/documents/562958574623716/4536952228/1" length="1522" stream_1_length="1522"/>
    <atom:id>mid:10e6c65a4@xios.xcerion.com</atom:id>
    <dc:folder>562958574623716</dc:folder>
    <dc:document>4536952228</dc:document>
    <dc:root xmlns:dc="http://xcerion.com/directory.xsd">vcard</dc:root>
    <dc:firstname xmlns:dc="http://xcerion.com/directory.xsd">Karl</dc:firstname>
    <dc:lastname xmlns:dc="http://xcerion.com/directory.xsd">Hyltberg</dc:lastname>
    <dc:businessmail xmlns:dc="http://xcerion.com/directory.xsd">karl.hyltberg@xios3.com</dc:businessmail>
  </atom:entry>
</atom:feed>
```

Besides searching and filtering in data sets another common pattern is merging data sets for presentation. Multiple atom feeds can be combined and presented as a single view, which is extra useful when the different data sets are owned by different identities. Think of this as doing a join operation in a traditional relational database.

One example where this is very useful is in the Calendar application where not only different personal schedules can be combined into a single view, but schedules shared by other people can be integrated as well. This is done by allowing calendar layers represented as one container for each calendar layer, through the security and sharing capabilities of the data model (using Access Control Lists, identities, and sharing) to be shared across identities.

This means that the developer of a calendar application only have to design the data model with layers as containers, the XML application for a calendar event and then start storing XML documents in containers. If a layer is to be shared, the container for that layer is then simply shared with the identities or groups that should have access.

10. CloudTop - Combining Applications into a Desktop

Stepping outside of the individual applications there is a lot of extra infrastructure needed to make applications run together within the same runtime environment and browser tab. Assuming the actual execution, rendering, and sharing of resources are solved problems, to really unlock the individual application's potential they need to cooperate and be able to open in multiple instances similar to how several emails or Word documents can be opened in a desktop operating system.

There are at least three areas of cross-application integration when we have isolated to a single system: datatype controls of actions, moving of data through user actions, and real-time sharing of resources between running applications and across different clients and identities.

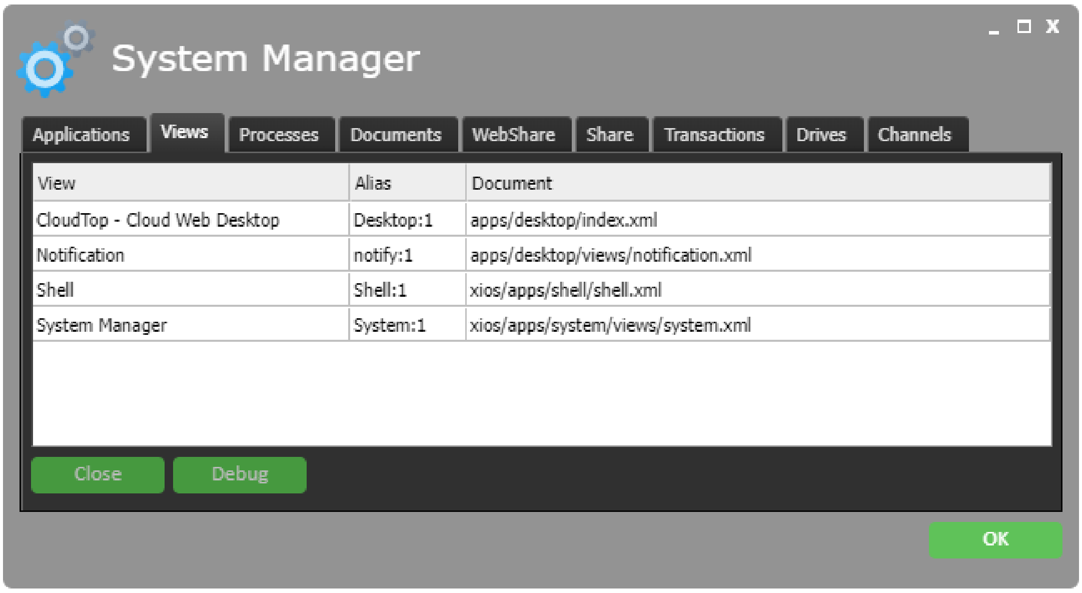
Datatype-based actions are commonly implemented in operating systems as a filetype register, where based on some meta-data property such as filename extension or mime-type property, the system could find the appropriate file icon and associated application to open it. We have already described how the file type-based meta-data extraction of properties works, but the datatype manager can do more. A file type, besides the user presentation data like description and icon, contains rules for how to identify a file type. For XML that is through one or more of namespace, root node, file extension, or mime-type. The definition can also define default applications for actions like open, edit, and preview that the desktop application can call without knowing what application will take over.

That kind of handover is simple as the application is given references to full documents that it is pre-arranged to handle. If however the user decides to drag-and-drop something or copy-and-paste it the application must be able to handle subtrees or fragments of documents. The approach XIOS/3 and CloudBackend has opted for is that all data within the system is handled by reference. Every document has a unique URL. To that we add an XPath to denote the relative root, useful when documents are inlined in other documents, and a set of XPaths that represents a selection within the document. Thus a drop or a paste action in a UI component in one application is just a reference to the same data as in the first application. This have some similarities to the XPointer and XPath standard, but combines a base selection with additional selections relative to the base selection. Think, a file list and selecting a couple of files.

Finally we have the real-time sharing of resources enabling inter application collaboration, but also collaboration between users across multiple clients, but with access to the same XML document in the data model. With the abstracted data model and security model, it becomes straightforward and ties into the architecture of intelligent UI components of XIOS/3. Since all applications are accessing data by reference, the built-in XML transaction manager just looks at all incoming changes and sends out change notifications to applications and any listeners in the cloud, if it happens to observe that a node is subject to change.

All internal bookkeeping and state is of course happening through XML documents, which themselves are subject to inspection. The System Manager application simply binds the various internal documents, like process list, application list, document cache, and transaction log, to grid UI components that display the content as tables. Things become a lot easier when everything is XML.

Figure 4. The System Manager.



View	Alias	Document
CloudTop - Cloud Web Desktop	Desktop:1	apps/desktop/index.xml
Notification	notify:1	apps/desktop/views/notification.xml
Shell	Shell:1	xios/apps/shell/shell.xml
System Manager	System:1	xios/apps/system/views/system.xml

Markup UK

Using TDD to produce High Quality XSLT

Return on experience

Christophe Marchand, Oxiane

This paper explains the benefits we had in using TDD to developp a Markdown to XML transformer.

It introduces TDD, Clean Code and Refactoring, and shows how to apply them to XSLT language.

At OXiane, we produce a lot of courses, we sell and teach. We were using office tools, like LibreOffice Writer and LibreOffice Impress. But it was difficult for us to make a clear distinction between a source document, that can be edited, and a production document, that is archived to know which slides has been used for which session. And being many writers to work together is not easy with such tools.

For this reasons, and for editorial contraints, we decided that a Oxiane Course will be a project, that contains sources for slides, source for exercise book, where sources are text files, and that we will build the production files by a build process.

This drives us to write various tools to transform text source files into PDF or HTML outputs. And we wanted to have high quality tools, so we decided to apply lessons learn from Software Craftsmanship.

1. TDD : where does it comes from ?

Test Driven Development was originaly introduced by Kent Beck in eXtreme Programming explained[xp-explained] in 2004, but was already used since the beginning of extreme programming in the end of 90's.

Extreme programming was a set of responses to the high costs, poor quality, unpredictable delays of software, and TDD focuses on quality. Quality is defined by readability, robustness and maintanability.

1.1. The TDD loop and the refactoring phase

Test Driven Development consists in first writing a test that describes a business requirement ; this test, when executed, must fail. Then, it consists in writing the minimum of code that make the test succeed. At this point, the business requirement described by the test is fully implemented, and business code can be delivered. But probably, the code isn't readable, isn't robust, has some duplications, and must be refactored to remove all duplications, to be easily readable and understandable, and to become robust.

This process is known as the TDD loop. And the most important phase of this loop is the refactoring phase. It's in this phase that we introduce methods that hides the technical complexity, and that describes the business intent. Once the business intent is clearly exposed in code, the code is readable, understandable, and code is easy to modify.

Refactoring is a process where we have to rewrite the code without changing any behavior. The sole way to guarantee this is to have unit tests, and to run unit tests each time we modify code, and to check that all tests succeed. If tests still succeed, program behavior has not been changed.

2. Writing a MarkDown to HTML converter with XSLT

The business purpose is to be able to write OXiane exercices books with only pure text. We do not want anymore any office tool, as they use binary file formats. These binary file formats are not fully supported by version control tools, such as Git. Ans all course writers at OXiane are developers that use Git every day.

So we need a pure text file format, easy to write and to structurate, and we need tools to transform this text-based file format to various outputs : LibreOffice for backward compatibility, Microsoft Office because some of our clients expect this file format, PDF for printers, and HTML for students - that's much comfortable for them to use than PDF.

The semantics requirements are very limited : paragraphs, three levels of titles, two levels of list-items, pictures, code blocks, bold text, inline code, links and anchors, and bold text in code blocks.

As MarkDown is commonly used in tools such as GitLab, we decided to use a limited subset of MarkDown, with special extensions to have bold text in code blocks. But it is really a limited subset, and we do not want to allow any writer to use some editorial features not allowed in our editorial rules. So, using an existing MarkDown converter was not a solution. We decided so to write our own MarkDown converter.

2.1. Feature definition

A course is a git repository. It contains a slide-deck, edited by an OXiane tool, and serialized in JSon, and this tool is not related to this paper. And it contains an exercice book, edited by any text editor in our own MarkDown flavor. It is also a project that is build by a continuous integration process, and this build process must produce the slide-deck in pdf format, and the exercice book in PDF and Html formats. LibreOffice and MS Office formats has been made obsolete. For both slide-deck and exercice book, two flavors are produced, one for screen, one for print.

We need a tool that is able to transform MarkDown to Html, to PDF, and a What-You-See-Is-What-You-Get browser-based editor. Transformation process must be quick and efficient, as WYSISWYG editor must be fluid. As this tool is written by a course-writers team, and as there is no credit to maintain this tool, it must be very robust, it must be very easy to evolve if new features are required in the future. It must provide comprehensive error message when transformation fails, to help the writer to correct its document, without requiring a developer assistance.

2.2. Tooling

As XSLT is able to run embeded in a Java Program, and in a browser, we choose XSLT to write the transformer program. Because we've used Saxon for a while, Saxon XSLT processor will be used.

XSpec is a Unit Test framework, that can be used to test XSLT, XQuery and Schematron. XSpec has been present for a while, but is actively maintained by AirQuick and Galtm

OXygen XML Developer[Oxygen XML Developer] is a really convinient Development environment for XSLT, it embeds Saxon, and XSpec. We will use OXygen XML Developer to develop this tool.

2.3. Methods

As Software Craftsmanship practitioners, we will use Test Driven Development to develop our transformer. Even it's not commonly used with XSLT, it has proven that if correctly practised, it produces high quality code.

3. Implementation

3.1. Level 1 titles

We want to transform a `# Title 1` line into an XML element `<title>Title 1</title>`.

Test Driven Development says we *must* write a test that describes the business requirement, before coding any production code. XSpec is a framework where we can describe business requirements as scenarios, with context and expectations.

Let's write such a scenario

Example 1. XSpec 1

```
<x:description
  xmlns:x="http://www.jenitennison.com/xslt/xspec"
  stylesheet="../../main/xsl/md-to-xml.xsl"
  xslt-version="3.0">

  <x:scenario label="Title 1">
    <x:context select="'# Title 1'" mode="convert"/>
    <x:expect label="A title element with text in">
      <title>Title 1</title>
    </x:expect>
  </x:scenario>
</x:description>
```

If we run this XSpec scenario, it fails, and says that `../../main/xsl/md-to-xml.xsl does not exist`. Correct. A test that does not compile is a failing test. We are allowed to write production code to make the test compile.

So we create the missing file, with the minimum code to make the test compile.

Example 2. XSLT 1

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0">

</xsl:stylesheet>
```

With this file, the unit tests executes, but fails to succeed. We can now write the minimum code to make the test succeed :

Example 3. XSLT 2

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0">
  <xsl:template match=".[. eq '# Title 1']" mode="convert">
    <title>Title 1</title>
  </xsl:template>
</xsl:stylesheet>
```

If we run again the unit test, it now succeeds. First part of TDD loop is completed, the code behaves as expected. But the code is very specific, and not very easy to understand ; business intention can not be quickly understood, and that's a problem. In [Clean Code], Bob Martin explains that we have to improve readability, it makes code easier to maintain.

To refactor code, we can process by hand, and apply manually various refactoring operations, like Extract Function[Refactoring Extract Function] ; we can also use OXygen's implementation of this operation, it will be quicker :

- Select `. eq '#Title 1'`
- Right Click on it / Refactoring / Extract Function
- It asks for a function name, let's type `prv:isTitle1` and validate
- OXygen generates the function, defines the namespace alias, and replace the ugly code by the function call :

Example 4. XSLT 3

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0"
  xmlns:prv="http://www.example.com/fn">

  <xsl:function name="prv:isTitle1">
    <xsl:value-of select=". eq '# Title 1'"/>
  </xsl:function>
  <xsl:template match=".[prv:isTitle1()]" mode="convert">
    <title>Title 1</title>
  </xsl:template>
</xsl:stylesheet>
```

Let's run again the unit test, it fails. The function does not have any parameter, no context item, so it can not compare text. Let's introduce a parameter ; OXygen does not provide a Introduce parameter[Refactoring Introduce Parameter] nor a Change function declaration[Refactoring Change Function Declaration], so we have to do it manually :

Example 5. XSLT 4

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0"
  xmlns:prv="http://www.example.com/fn"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:function name="prv:isTitle1">
    <xsl:param name="line" as="xs:string"/>
    <xsl:value-of select="$line eq '# Title 1'"/>
  </xsl:function>
  <xsl:template match=".[prv:isTitle1(.)]" mode="convert">
```

```
    <title>Title 1</title>
  </xsl:template>
</xsl:stylesheet>
```

The test succeeds. But function is at the beginning of the code, and it hides the business part. Let's apply the Slide Statement[Refactoring Slide Statement] refactoring operation, to move the function at the bottom of the file :

Example 6. XSLT 5

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0"
  xmlns:prv="http://www.example.com/fn"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:template match=".[prv:isTitle1(.)]" mode="convert">
    <title>Title 1</title>
  </xsl:template>

  <xsl:function name="prv:isTitle1">
    <xsl:param name="line" as="xs:string"/>
    <xsl:value-of select="$line eq '# Title 1'"/>
  </xsl:function>
</xsl:stylesheet>
```

Now, we can read the XSL file, and understand that it is used to convert all lines that are a `title1`, and transform it to a random element. We do not need to know exactly what is a `title1`, and which technical rules apply to identify a `title1`. We just know that this template processes all `title1` lines.

This second part of TDD loop, the refactoring part, is the most important phase of TDD. It's the phase where we make business rules emerge from the code.

What a pitty, Oxygen is not very precise in refactoring, and it generates code that has to be manually fixed. The generated function should be typed as boolean, but isn't. It's an error, and we know this because we have good skills in XSLT. But we can not write new code without a failing test. So we write a test that shows the function is not correct and must return a `xs:boolean`.

Example 7. XSpec 2

```
<x:description
  xmlns:x="http://www.jenitennison.com/xslt/xspec"
  xmlns:prv="http://www.example.com/fn"
  stylesheet="../../main/xsl/md-to-xml.xsl"
  xslt-version="3.0">

  <x:scenario label="Title 1">
    <x:context select="'# Title 1'" mode="convert"/>
    <x:expect label="A title element with text in">
      <title>Title 1</title>
    </x:expect>
  </x:scenario>

  <x:scenario label="testing isTitle1">
```

```
    <x:scenario label="# Title 1">
      <x:call function="prv:isTitle1">
        <x:param># Title 1</x:param>
      </x:call>
      <x:expect label="Title 1" select="true()"/>
    </x:scenario>
  </x:scenario>
</x:description>
```

Here, we precisely define we expect a `xs:boolean`. We we run the test, it fails. Types are different. Let's correct production code just add `as` attribute :

Example 8. XSLT 6

```
  <xsl:function name="prv:isTitle1" as="xs:boolean">
    <xsl:param name="line" as="xs:string"/>
    <xsl:value-of select="$line eq '# Title 1'"/>
  </xsl:function>
```

And here, Oxygen shows a warning on mis-use of value-of. We replace it with a sequence :

Example 9. XSLT 7

```
  <xsl:function name="prv:isTitle1" as="xs:boolean">
    <xsl:param name="line" as="xs:string"/>
    <xsl:sequence select="$line eq '# Title 1'"/>
  </xsl:function>
```

Now, all tests are green.

We now can add new test cases for titles. XSpec allows to wrap scenarios into scenarios, it is very useful to group scenarios together ; that's what we do here, to check if our `isTitle1` function is correct :

Example 10. XSpec 3

```
  <x:scenario label="testing isTitle1">
    <x:scenario label="# Title 1">
      <x:call function="prv:isTitle1">
        <x:param># Title 1</x:param>
      </x:call>
      <x:expect label="true" select="true()"/>
    </x:scenario>
    <x:scenario label="# Another title 1">
      <x:call function="prv:isTitle1">
        <x:param># Another title 1</x:param>
      </x:call>
      <x:expect label="true" select="true()"/>
    </x:scenario>
  </x:scenario>
```

The new scenario fails, our function is not correctly written. As we have read the Markdown specification, we know that a level 1 title must starts with `# `, so we code this :

Example 11. XSLT 8

```
  <xsl:function name="prv:isTitle1" as="xs:boolean">
    <xsl:param name="line" as="xs:string"/>
    <xsl:sequence select="$line => starts-with('# ')"/>
  </xsl:function>
```

And all tests succeed.

Last, we have to test that the produced `title` element contains the correct text. So we write a new scenario with a different text. And we refactor scenarios titles, to be clearer :

Example 12. XSpec 4

```
<x:scenario label="Exercise Book title">
  <x:scenario label="Title 1">
    <x:context select="'# Title 1'" mode="convert"/>
    <x:expect label="A title element with text in">
      <title>Title 1</title>
    </x:expect>
  </x:scenario>
  <x:scenario label="Another title 1">
    <x:context select="'# Another title 1'" mode="convert"/>
    <x:expect label="A title element with Another title 1 in">
      <title>Another title 1</title>
    </x:expect>
  </x:scenario>
</x:scenario>
```

This new scenario fails, we can correct the template to produce the expected output :

Example 13. XSLT 9

```
<xsl:template match=".[prv:isTitle1(.)]" mode="convert" expand-text="true">
  <title>{. => substring(3)}</title>
</xsl:template>
```

This new code make all scenarios succeed. We are done with implementation for level 1 titles, but code is not very understandable : what does `. => substring(3)` mean from a business point of view ? Nothing, it's too technical. Let's add clarity, and extract all the thecnical code in a function :

Example 14. XSLT 10

```
<xsl:function name="prv:getTitle1Content">
  <xsl:param name="line" as="xs:string"/>
  <xsl:value-of select="$line => substring(3)"/>
</xsl:function>
<xsl:template match=".[prv:isTitle1(.)]" mode="convert" expand-text="true">
  <title>{prv:getTitle1Content(.)}</title>
</xsl:template>
```

And again, move function to the bottom of file, it's only technical details... Now, if we look at our code, we have a template where the match attribute contains an understable business rule, and it produces an output that is also described by a business rule. All rules are clearly defined in the code. This code will be much simpler to maintain if business rules evolve in the future, and if developers who write the initial code are not still active on the project. We can read the code from top to bottom, as a book.

Example 15. XSLT 11

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="3.0"
  xmlns:prv="com:oxiane:courses:mdtoxml:private"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xsl:mode name="convert" on-no-match="fail"/>

  <xsl:template match=".[prv:isTitle1(.)]" mode="convert" expand-text="true">
    <title>{prv:getTitle1Content(.)}</title>
  </xsl:template>

  <xsl:function name="prv:isTitle1" as="xs:boolean">
    <xsl:param name="line" as="xs:string"/>
    <xsl:sequence select="$line => starts-with('# ')"/>
  </xsl:function>

  <xsl:function name="prv:getTitle1Content">
    <xsl:param name="line" as="xs:string"/>
    <xsl:value-of select="$line => substring(3)"/>
  </xsl:function>
</xsl:stylesheet>
```

3.2. Next titles and list items

Then, the same process is applied to level 2 and level 3 titles, to level 1 and level 2 list items. We write a new test case, we check that the test fail, we write the minimum code to make the test succeed, and we refactor production code to make it readable.

4. Pro and Cons of using TDD for XSLT development

When applying strictly Test Driven Development method, we only write code that is required by a business use case. we do not have any unnecessary code.

When appying with test driven development, developers proceed by baby steps. This allows to focus on a simple problem, and produces simple code.

Refactoring is the phase where code is reorganised. It eliminates duplication, and make appears design that simplifies code. Code is simplified as features are added[Ward Cunningham]

Extracting code to functions allows the business requirement being exposed clearly. It also conduct to respect the Single Responsability Principle. Respecting the SRP makes the code more robust and more easy to maintain.

As all business requirements are exposed as unit tests, and as the tests all succeed, there is no bug in code.

As we write only code that is required to make a test succeed, all code is covered by unit tests. TDD drives us to have a 100% code coverage ; it's difficult to have such a coverage with other methods.

Data coverage is more a problem. We can not write code to process unexpected or unknown data, as we are not able to write test cases for this unexpected data ; but we can write code that fails if there is an unexpected data structure.

4.1. Bugs

When a program do not behave as expected, we call this a bug. Using TDD forces to express all expected behavior by tests. So all behavior is perfectly known and proven. If TDD is used, then there is no bug in our code.

We may have new expectations. A client may ask for a new semantic, a new rendering. It's not a bug. It has not been expressed before, we have to implement a new behavior. We will do this by applying TDD. Creating new tests, implementing code to make tests succeed may drives to have existing tests that fail. We have to check carefully if we broke the expected behavior, or if the existing test that fail is still valid. Sometime, adding new features drives to change existing features, and so drives to change tests.

On the OXiane Course project, we do not have any ticket that was an error, after investigation. We have tickets where the client says "it doesn't work as I expect". But it works as it was expected in the beginning. And these tickets have conduct us to change some parts of syntax, mainly on image sizing.

4.2. Baby steps

Coding with TDD, we should proceed by baby steps. Only small things, one by one. When this first thing works, then you create a second test, with a second baby step, and you code it.

This allows the developer to focus on problems one by one. And not to disperse in many problems.

When a bay step is coded wand when all tests succeed, we are able to deliver our code. It may be complete, but everything that's coded works, is documented by steps, and is reliable. We are able to deliver the final product to final users, and they are able to evaluate it. This ability to deliver all the time is required by agile methods.

This may conduct to have a lot of corks in our production code, if developer can not find the the way to make the code right (Make it run, make it correct, make it fast[xp-explained]). But with this corks, the code behaves correctly, according to written unit tests. The corks will disapear in refactoring phase.

4.3. Refactoring

As written above, refactoring is the most important phase of TDD loop. This experience of using TDD with XSLT has shown that refactoring tools provided by Oxygen are limited, and they can be enhanced to provide high level helpers. Other IDE, like IntelliJ or Eclipse, provide very advanced refactoring tools, and this tools allows developpers to be much more efficient. For example, when we rename a function, all code that calls this function is updated to take in account the rename. But Oxygen has already this refactoring operations, some implemented in editor, like move XML element, which fits with slide statement[Refactoring Slide Statement], some from the Refactoring menu. Users probably need to challenge Oxygen team to enhance these refactoring operations.

Refactoring is the phase where design emerges. When we see many corks written one after the other, we see what we have to do to introduce the right design, or the right patterns to eliminate duplications and to simplify the code.

4.4. Code Coverage

Code coverage is a common indicator of code quality. Tools like Sonarqube have metrics on code coverage. Code coverage, in object oriented languages, is calculated on lines of code. But in these languages, there is mainly only one instruction per line of code.

In XSLT, spacing in code is part of code. If we separate two sequence constructors by a space or a carriage return, the output is changed. And so we may have more than one instruction per line of code. Calculating code coverage with XSLT is more complex than with other languages.

XSpec is able to calculate code coverage with XSLT. It requires to use Saxon-EE. XSpec does not provide a numeric result for code coverage, but there is a colored rendering of XSLT code that allows to know if a code instruction is tested or not. Since Saxon 10, API has changed and XSpec framework has not been fully updated for these API changes. Michael Kay (Saxonica) has contributed to XSpec framework to correct this, but there is still some job to do to have good results on code coverage evaluation. If we keep a line scope for coverage, and if we are able to ignore XML vocabulary like namespace declarations, then we can see that all the XSLT code is covered by unit tests.

4.5. Data Coverage

Data coverage is another important quality indicator. We have to be sure that our code covers all data cases. As XSLT is a data-driven process, there are some mechanisms in XSLT to ensure that all data cases are processed, like `no-match` attributes in `xsl:mode`. But it is sometime not enough, as we can decide to process only parts of input content, with `xsl:apply-templates/@select` for examples.

A common way to ensure that all data cases are covered is to use grammars on input content. XML Schema[XML Schema], RelaxNG[RelaxNG] and Schematron[Schematron] are commons grammars used with XML content, and at least XML Schema and Schematron can easily be embeded in an XSLT transformation. But in our case, input is pure text, with no markup, and these grammars are not made to validate text.

Today, we are only 6 people to use this tool, and we decided to ignore data coverage problems. If a writer use an unsupported syntax, it is simply processed as simple text, and sent to output without any transform. It is neither satisfying nor robust, but, at this point of usability, we will not invest more in this part.

5. Conclusion

Test Driven Development, Refactoring, Clean Code and Software Craftsmanship are all techniques and methods that produces high quality code in Oriented-Object Programming languages.

XSpec framework, and its integration in Oxygen, allows to apply test driven development with XSLT. It runs quickly enough to have a short TDD loop, and to produce quickly deliverable code.

Coding time is a little it more important than with standard method, but it produces code that is much cheaper to maintain, and much more reliable.

Mixing TDD and XSLT provides a lot of serinity to developers.

Bibliography

[xp-explained] Kent Beck: eXtreme Programming explained. ADDISON-WESLEY: 0-201-61641-6

[Clean Code] Robert C. Martin: Clean Code. Addison-Wesley: 978-0-13-235088-4

[Refactoring Extract Function] Martin Fowler: Refactoring, Second Edition Addison Wesley: 978-0-13-475759-9 ; Extract Function - Page 106

[Refactoring Change Function Declaration] Martin Fowler: Refactoring, Second Edition Addison Wesley: 978-0-13-475759-9 ; Change Function Declaration - Page 124

[Refactoring Introduce Parameter] Martin Fowler: Refactoring, Second Edition Addison Wesley: 978-0-13-475759-9 ; Introduce Parameter Object - Page 140

[Refactoring Slide Statement] Martin Fowler: Refactoring, Second Edition Addison Wesley: 978-0-13-475759-9 ; Slide Statement - Page 223

[Oxygen XML Developer] SynchRO Soft SRL https://www.oxygenxml.com/xml_developer.html

[Ward Cunningham] Ward Cunningham: The WyCash Portfolio Management System, Experience Report https://dl.acm.org/doi/pdf/10.1145/157709.157715 Page 2/2, end of first paragraph

[XML Schema] Shudi Gao C. M. Sperberg-McQueen Henry S. Thompson Noah Mendelsohn David Beech Murray Maloney : W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures https://www.w3.org/TR/xmlschema11-1/

[RelaxNG] James Clarck MURATA Makoto : RELAX NG Specification https://relaxng.org/spec-20011203.html

[Schematron] Document Schema Definition Language (DSDL) - Part 3: Rule-based validation using Schematron https://www.iso.org/obp/ui/#iso:std:iso-iec:19757:-3:ed-3:v1:en

Word processing is so last century

Formalizing internal narratives using internal declarations and making them look pretty

Kurt Conrad, The Sagebrush Group

This presentation describes a journey of replacing WordPerfect with prodoc.dtd, a semantic authoring doctype; and how prodoc evolved to enable computer-assisted sense making, based on markup that formalizes knowledge flow analysis and modeling semantics.

The paper also describes some of the challenges associated with multi-perspective decision making and techniques for negotiating and formalizing dynamic, natural-system ontologies. Lessons learned from WordNet and SUMO integrations are summarized.

The paper concludes by suggesting that building editors and bots that use author-authored markup to digitize and amplify logic systems could, in many organizational settings, contribute to more-intelligent value optimizations and better long-term performance.

1. Executive Summary

Sense making is all about knowing what to do in a given situation, both individually and collectively. Many observe problems with our capacity for collaborative, intelligent behavior. As automated systems (bots) get more involved in decision making, the world must be described more explicitly for intelligent, automated behavior.

This is where natural-system and formalized ontologies come into play.

This paper focuses on using markup to capture the natural languages that individuals use when they make sense of the world — the languages of internal narratives — to start negotiations around formalizing natural system ontologies into automated support systems to help individuals and groups make sense of complexity from multiple perspectives.

Formalizing an individual's personal ontology involves digitizing the way that they think about and organize information. This paper reports lessons learned from experiments developing a personalized XML doctype, prodoc. After being modified, as needed, for over a decade, prodoc has evolved to ease the authoring of information from many different perspectives/ logical systems/ ontologies.

> What makes for a better bot? One that's fast, cheap, low-risk, easily trained, and thinks like me.
>
> —Test subject

The first section, *What's computer-assisted sense making?* looks at the role of automation in helping see patterns and organizing actionable information to enable the intelligent behavior of individuals, groups, and bots. It starts by relating semantic markup, semantic technologies, and formalized ontologies to human and automated sense making processes.

The path to prodoc contrasts word processing's focus on visual formatting behaviors and storage models with generalized markup, where text files contain start and end tags that describe nested containers that make content easy to process and stylesheets provide visual formatting specifications.

It introduces semantic markup, which involves giving the containers meaningful names, and semantic authoring, which allows authors to easily create new markup and name the data structures, themselves. Technically simple, allowing authors to declare new markup within documents has significant policy and value-proposition impacts.

prodoc in practice describes computer-assisted sense making tools that a test subject developed using custom markup and element/ attribute-based control surfaces. These control surfaces, sometimes augmented with form controls, allow the author to easily change and adjust CSS visual rendering properties. Sheet music, engineering accessible color pallets, and a modeling language are described.

The politics of markup and meaning deals with politics, which is defined as the way that groups make decisions. Complex global publishing lifecycles bring countless perspectives to the negotiation table. When semantic formalization and authoring are viewed from this perspective, semantics get dynamic. People change their minds about what's meaningful and how to communicate it.

A generalized process for values-based decision making is introduced, and its use in stabilizing and formalizing ontologies during markup development is described. Possible roles for semantic authoring in bottom-up data negotiations are considered. The results from mapping personalized markup to authoritative third-parties, WordNet and SUMO, are reported.

The paper concludes by introducing H1, a semantic authoring training system that is available for testing.

2. What's computer-assisted sense making?

Sense making is all about knowing what to do in a given situation, both individually and collectively. Many observe problems with our capacity for collaborative, intelligent behavior.

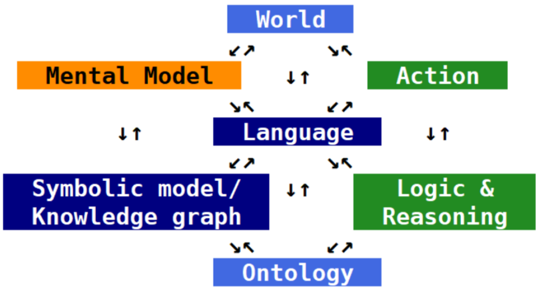
Computers can assist with communications. Likewise, they can assist with sense making, helping people organize information, visualize information, see patterns, and make decisions. As automated systems (bots) get more involved in decision making, the world must be described more explicitly for intelligent, automated behavior.

2.1. Onto-what?

What does it take to explain everything to a computer? This is where the ways that we perceive existence (natural-system ontologies) and formalized computing ontologies come into play. What's an ontology? Here are a couple of descriptions:

> Ontology is the subject that asks the question “What is there?”
>
> The answer can be stated in one word: “Everything.”
>
> —W. V. O. Quine

Figure 1. Sowa Hexagon



John F. Sowa's Hexagon puts language at the center and emphasizes the importance of mental models and logic in the formalization of an ontology. Formalizing real-world and abstract concepts for our buddies the bots effectively involves not only dealing with “what exists?” as a list of terms, but also by dealing with all of the models and behaviors that Sowa identifies.

Fully-formalized computing ontologies associate terms with formal logic. That's really expensive. Markup, from this perspective, could be considered a semi-formalized ontology.

Figure 2. Formalized ontologies

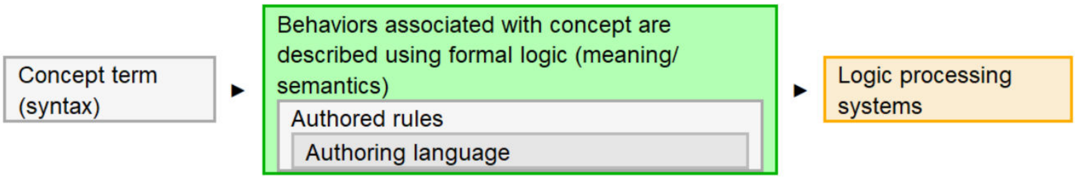
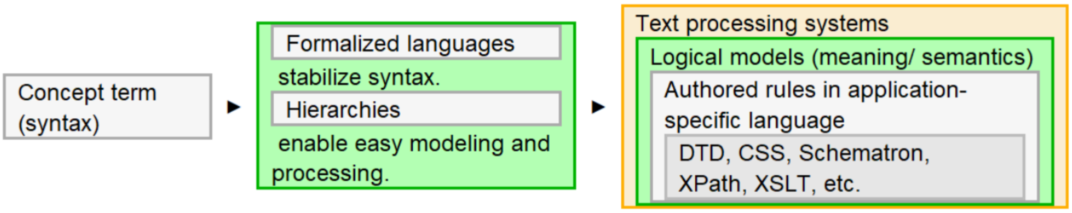
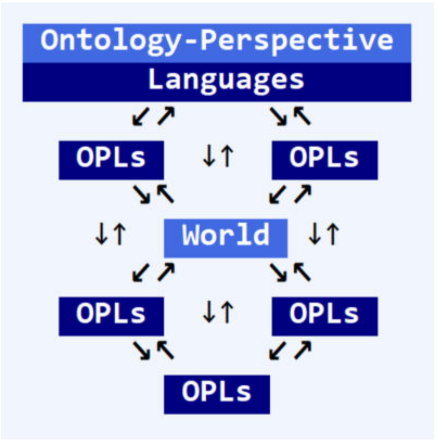

Concept term (syntax)
Behaviors associated with concept are described using formal logic (meaning/ semantics)
Authored rules
Authoring language
Logic processing systems

Figure 3. Semi-formalized, markup-based ontologies


Concept term (syntax)
Formalized languages
stabilize syntax.
Hierarchies
enable easy modeling and processing.
Text processing systems
Logical models (meaning/ semantics)
Authored rules in application-specific language
DTD, CSS, Schematron, XPath, XSLT, etc.

2.2. Working with multiple languages, perspectives, and ontologies

Figure 4. Conrad corollary



The Conrad corollary to Sowa's hexagon sees the world awash in languages that reflect different ontological perspectives, concepts, and logic systems. These are reflected in terminology and usage. Some are naturally-occurring. Some are engineered. Some are dysfunctional. Modern communication systems allow them to compete, globally, for attention and influence.

attempted to integrate and synthesize multiple ontologies and associated languages:

- Word processing and its visual concept of the document, which focuses on how text is arranged and looks on a page, and the ad-hoc, often-redundant behaviors associated with applying layout and style formatting rules with a mouse.

- The UNIX tradition of little languages and little bots for text processing.

- Semantic markup, SGML, XML, etc., which are based on IBM's hierarchical concept of the document.

- Knowledge management and engineering. Concepts for describing the fractal ways that data is transformed into information and synthesized into actionable knowledge to enable intelligent, intentional behavior. Data's about the details. Information's about organizing. Knowledge's about actionable behaviors.

- Multiple operational ontologies for concepts of semantics and meaning. What “meaning” means for humans, automated, and organizational systems. How semantic technologies and formalized ontologies can be used to engineer knowledge about and understanding of meaning in these different behavioral contexts..

- Policy making and performance. Techniques for helping folks articulate perceived values to rapidly reach agreements around terms, meanings, priorities, and operational details. The vocabularies of the Government Performance and Results Act, activity-based costing, and value canvasses.

> Remember, style is something that things fall out of, and relational database tables are OK for data, but didn't **and don't** work for organizing documents.
>
> —Test Subject

2.3. How prodoc helped make sense of big words

The test subject's experiments in computer-assisted sense making involved using markup to

- Combine concepts and techniques to model associations

- Analyze and refine those models to design and debug knowledge flows and computing systems

- Develop ad-hoc and switchable visualizations to identify, highlight, and communicate informative patterns

- Iteratively-refine markup and support systems based on new priorities and learnings

- Negotiate “optimal” balances of operational and management considerations, such as priorities and quality criteria, author effort, development effort, markup flexibility and specificity, cool examples, schedule, lifecycle costs, and performance

- Build agile, continuous delivery pipelines that enable anything, anytime, modifications to structure, layout, look, and feel; producing highly-contextualized PDFs and invalid, funny-looking source documents.

2.4. Lessons learned

Co-locating knowledge and behavior tends to improve performance. Multiple perspectives tend to improve quality (and avoid crashes). The big challenge is how to integrate multiple perspectives and synthesize holistic, intelligent behavior.

Markup brings important capabilities to the table. It helps with the mechanics of getting knowledge to the agents responsible for behavior. It's both human and machine readable, and makes for an excellent negotiation framework.

Knowledge gaps to be identified and resolved through analysis of knowledge flows and language patterns. Plugging knowledge gaps better align behaviors and improve overall performance. Knowledge flows can be engineered to enable specific organizational behaviors and disable others, either through knowledge gaps or cost drivers. That's quite the ticket when associated with behavior prediction markets.

Shared meaning is negotiated. Meaning has multiple dimensions. When multiple agents interact with knowledge, they bring multiple perspectives. Just for starters, there's the plumbing side, how to move knowledge between agents.

Adding the policy and performance perspectives brings, “Why inform them?”“Who benefits?”“What are the costs?” The list of unique stakeholder perspectives can be countless, especially after folks apply new logic systems after changing their minds.

Knowledge architectures relate the bits and pieces of how knowledge enables behavior. Markup can be used in a knowledge architecture not only to enforce top down data quality standards, but enable the bottom-up negotiation of complex systems.

One way to use markup to integrate multiple-perspectives is by capturing the natural languages that individuals use when making sense of the world — the languages of internal narratives — and using those languages to start negotiations around formalizing natural system ontologies into automated support systems (to improve ergonomics and operational performance) and establishing community standards (for management system performance).

Formalizing an individual's personal ontology involves digitizing the way that they think about and organize information. This paper reports lessons learned from experiments developing a personalized XML doctype, prodoc. After being modified, as needed, for over a decade, prodoc has evolved to ease the authoring of information from many different perspectives/ logical systems/ ontologies.

The development of specialized data structures is at the essence of the idea of computer-assisted sense making. Organizing information, both structurally and visually, enables patterns to be identified and logic systems applied to enable behavior. When the situation makes sense, you know what to do.

Bots can do many things to help things make sense. Many experiments involved fine tuning the visual characteristics of authoring interfaces to make authoring as ergonomically-comfortable and time-efficient as possible.

At times, multi-perspective visualizations conflict with each other, potentially damaging signal-to-noise ratios. Switches are useful to help visualize patterns, even shifting layouts and visual style mappings to highlight different aspects, as appropriate. Explicitly-structured markup makes these types of processing and rendering tricks much easier to operationalize.

3. The path to prodoc

prodoc is the synthesis of:

- ◇ Some relatively simple things (e.g., word processing)
- ◇ Some surprisingly simple things (e.g., helping people figure out what the agree on)
- ◇ Some medium complexity things (e.g., markup systems)
- ◇ And some somewhat more-complex things (e.g., understanding knowledge flows from multiple perspectives)

> “No, prodoc's not available. It's an ecosystem of polished production systems and abandoned experiments. More importantly, you'd have to think like me. That process would likely hurt both of us.”
>
> —Test subject

3.1. Bots hate word processing's ornery visually-oriented ontology

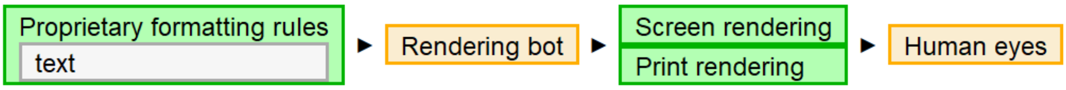
Let's start with the familiar. Imagine building a spreadsheet with only three columns: **bold**, *italic*, and underline? How would you make sense of anything?

Word processing (WP) is an approach to documents that is firmly planted in the world of paper. It started by managing whitespace and adding a few visual formatting tricks to replace typewriters. Apple dramatically focused attention on defining the document in terms of its visual characteristics when it introduced the Macintosh, with synchronized bitmapped displays and printers.

We're still largely at that stage, where the bulk of the world's documentation is encoded in proprietary formats that concern themselves mostly with visual formatting of information on physical and virtual pages. Even when the files are encoded using marked-up text, the metadata and other metaknowledge artifacts store mostly visual properties.

WP's visual orientation is problematic. It really requires humans to be in the loop to look, see, recognize, interpret, synthesize with other understandings to apply missing implicit knowledge, decide, and act.

Figure 5. Word processing



The visual metaphors also limit the ability of computers to both mediate human-to-human communications and also to participate in the conversation. Even the XML-based word processing encodings are rats nests of difficult-to-process markup. The noise of visual parameters swamps the signal of individual characters. Doable, but expensive.

And this is the bottom line of why WP is obsolete technology. It's just too expensive. Labor costs are too high, in large part because you can't build happy little bots to automate anything. You have to buy bots or hire someone to build big grumpy bots.

WP provides flexibility to easily tune appearance, but reuse? Global imperatives require fine-grained, context-sensitive documentation to drive intelligent behavior. That means automation. That means containers. Sure, you can move files, but you can't move fine-grained information in and out of narrative documents. WP data is, to quote someone I met at a non-destructive nuclear assay conference, “just too stupid to move.”

And this isn't the way that the big dogs roll. The publishing industry started walking away from this technology, decades ago.

3.2. Semantic markup to the rescue

In the 1990s professional publishers were getting cut up worse than the weak kid at a knife fight. Today, news publishers are really bleeding out.

The world of electronic documents was disrupting paper-based publishing. New terms, new technologies, and new processes forced new business logic and strategies. CD-ROMs hit first. Many organizations made significant investments to reengineer their documents to rapidly-release CD-ROMs.

Then the web hit, with its wicked learning. Publishers asked themselves, “How many times would this happen?”“How expensive would conversion costs be for each new technology cycle?”“How many different file formats will we need?”“How many different variations for different delivery devices?”

Fortunately, the web came with the seeds of the solution. The format used for web documents, the HyperText Markup Language (HTML), was based on an international standard, the Standard Generalized Markup Language (SGML). SGML defines rules for declaring, applying, and validating markup languages — a language for defining markup languages — a “meta-language” so to speak.

SGML was designed around the needs of document owners, authors, and publishers. It enables hub and spoke architectures to be implemented with authoritative, single sources of generalized markup that feed various delivery systems (CD-ROMs, web, etc.).

3.2.1. Separating structure and style

Fundamental to generalized markup is the separation of content from style. It leverages lessons learned that go back to John W. Seybold's work at ROCAPPI, where lookup tables were first used to replace complex and error-prone typesetting markup codes with short — easy to type and differentiate — strings of text.

Figure 6. Typesetting lookup table

Easy strings of text	Confusing, error-prone typesetting codes
"`<p>`"	;;; ldasj;fsadlkf dsalkf jdsa;l fjdsl;f dsl;f jdsafldslfkds ld :::
"`</p>`"	;;; a s;dfkjsadfklds fldskaj fdskf ds;lkf sdlkf dslf sadlfk jdsfkj ds :::

In this world, formatting is associated with generalized data structures, making it easy to apply new styles or transforms, as the need arises. For this document, it took a couple of days to:

- Add new data structures to prodoc to make authoring easier to match DocBook requirements.
- Create a new stylesheet, `pdoc2docbook.xsl`, to transform prodoc XML to DocBook XML.

  It stripped all of prodoc's attributes, mapped element names from one markup language to the other, and changed the structure of a few branches.
- Tune the XML source and XSLT files to create valid DocBook XML.

  prodoc structures that couldn't be rendered were converted to images.

Once the pipeline was stable, attention went back to authoring and editing. A handful of whitespace issues required manual cleanup.

3.2.2. Integrating lifecycle perspectives

The focus on document lifecycles drove a number of important features that are missing from contemporary word processing platforms and relational database systems.

Markup language files are text files. That simple fact improves accessibility and lowers lifecycle costs by orders of magnitude. Nearly any computing device and system can interact with a markup-encoded text file. The Extensible Markup Language (XML) dramatically lowered SGML's cost profile, and there are a wealth of XML-based markup technologies.

Documents are naturally hierarchical. Books contain chapters. Chapters contain paragraphs... Markup is organized around addressable trees and nodes and nested containers. These explicit data structures lower level-of-effort for both human and machine processing.

Semantics enter the picture when naming the nodes in the tree. When creating custom data structures to solve real-world authoring and publishing issues, the names are often associated with real-world meanings and organizing concepts. These terms, their meanings, and various encoding decisions take us into the world of semantic technologies and formalized ontologies.

Semantic markup is the phrase given to this approach to document management, where XML is used to define an application-specific markup language, where the names and data structures reflect complex conceptual frameworks and models.

The challenge when using markup for single sources is to focus all authoring on the single, highly-refined source document, so that all derivatives flow from there.

> Hub and spoke, baby, with low cost transforms. Hub and spoke.
>
> —Test subject

3.3. From semantic markup to semantic authoring

> prodoc helps me make sense of the world. Markup brings ergonomic efficiencies to the typing process. Semantic markup brings ergonomic efficiencies to the thinking process, especially when concepts and logic systems can be visualized in different ways.
>
> —Test subject

Most semantic markup systems are organized around the idea of collaborative authoring and publishing. An organization-specific markup language is defined as a doctype with a set of rules that describe the names and relationships of the various containers. Everyone in the group works within those rules, which function as quality standards.

Doctypes are designed to standardize documents and protect publishing systems. The structural rules, which are described using an explicit schema language, typically change very slowly to ensure consistency of work processes. Standardized stylesheets are usually part of the equation.

Semantic authoring inverts this thinking.

Authors are the semantic authorities, they should have all the operational authorizations available to author semantic markup as they deem appropriate, as easily as possible. Every doctype should allow authors to use the internal declaration subset to enable bottom-up formalization and communication of models, in addition to content.

Semantic authoring is about creating documents that help authors make sense of the world. It lets individuals create semantic markup to formalize their conceptual frameworks. Once formalized, these sense making systems are easier to reuse, communicate, and automate.

Reducing the incremental costs of new markup means document-level extensions, making every document a **little markup language laboratory** *The idea of “little languages” comes from AT&T's development of the UNIX operating system, where little languages and little bots were developed for managing the streams of text associated with managing telecommunication systems in the 1970s*.

Semantic authoring doctypes leverage the Internal Declaration Subset, which is a mechanism for authors to create their own markup declarations. This allows new elements and attributes to be declared within any document. They can even be used within the current document, if the rules of the doctype allow.

prodoc was built on an HTML core (HTML for semantic authoring: hsa.dtd), but comparable semantic authoring extensions could be incorporated into other doctypes. Adding semantic authoring extensions to a DITA subset have been discussed, for example.

Why open Pandora's Box? What chaos would ensue?

3.3.1. Individual impacts

Any number of strategic computing projects have failed because organizations couldn't get folks to use XML editors.

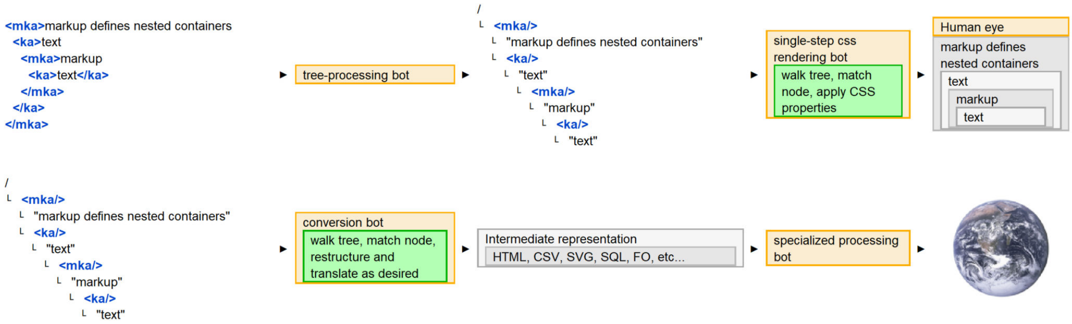
Why would an individual consider this craziness? Everyone's plate is full. This requires learning and ever since Microsoft took the margins out of WP through bounding, investments in this space have withered, for both individuals and organizations. WP is an ignored necessity, driven by cost-minimization strategies.

The standard XML consultant's answer to any question is, “That depends.” Markup is largely policy neutral. You can apply it to almost any set of value optimizations. Markup languages are adopted, customized, and designed around a purpose. When you use someone else's markup, you are organizing your information around someone else's semantic values and value propositions. That's a cost driver.

Everyone's plate is full because the world is increasingly complex. There are a lot of bots that help, but they're all pretty much built by someone else to help them meet their needs.

What if folks had bots of their own? Little bots that could be put together in an afternoon? Organized around personally-valuable information. Individuals creating their own little languages and bots inverts markup's dominant value propositions.

Figure 7. Generalized markup



```
<mka>markup defines nested containers
  <ka>text
    <mka>markup
      <ka>text</ka>
    </mka>
  </ka>
</mka>
```

```
/
└ <mka/>
  └ "markup defines nested containers"
  └ <ka/>
    └ "text"
      └ <mka/>
        └ "markup"
          └ <ka/>
            └ "text"
```

tree-processing bot
single-step css rendering bot
walk tree, match node, apply CSS properties
Human eye
markup defines nested containers
text
markup
text
conversion bot
walk tree, match node, restructure and translate as desired
Intermediate representation
HTML, CSV, SVG, SQL, FO, etc...
specialized processing bot

When concluding XML courses I challenge students to build their own “bootcamp app”:

◇ An XML fragment of information that has personal meaning.

◇ Some CSS to make it pretty and a DTD to validate it.

◇ Extra credit for an XSLT to HTML conversion. With that set of languages you can code the guts of almost any automated solution.

◇ Pro tip: Add a text processing language to get stuff into markup.

> “ AWK is awful good.”
>
> —Test subject

Reports ranged from ham radio datasets to a DVD collection with a color scheme that would drive a person to drink. The common denominator? Feelings of accomplishment and excitement about the future. Personal markup is compelling. It's the ultimate computer game.

When markup is optimized around individual purpose, it's an enabler, with corresponding changes to brain chemistry. A recent episode of the Public Broadcasting Series, Nova, focused on the neurochemistry of decision making.

> Agency is powerful stuff. Life is unscripted. Perceptions of control are powerful. Actual, impactful, personally-meaningful operational control? Priceless.
>
> —Test subject

The brain chemistry associated with fear is another important factor. Fearful adults staring at blank screens with their jobs on the line don't learn much. Fearful improvisers tend to get frantic. Open another document template and experiment with markup is like playdough and enables much more creativity.

> ...when people are made afraid, their amygdala starts firing and their prefrontal cortex literally is starved of blood. You can’t have both things [fear and logic] going. And so if you’re asking people to sort of be in their rational brain..., then you need to present this as the possibility of creating something good...
>
> —Anat Shenker-Osorio [https://slate.com/podcasts/amicus/2021/02/incitement-impeachment-inevitable-acquittal.amp]

> Future research hopes to work with neuro-divergent individuals, including those with sensory-integration impairments who have had to create very sophisticated coping mechanisms and supporting ontologies. How much of this knowledge is tacit and inexpressible? How much could be formalized as language and would there be benefits? Stories heal. Would this form of storytelling amplify and add additional dimensions to those benefits?
>
> —Primary researcher

Giving voice involves two dimensions, the content and the pipeline that communicates that voice. Authoring content that is designed for collaboration and integrates easily with existing computing infrastructures adds social dimensions to individual value propositions.

> Our voices are so valuable that we're the only animal that chokes itself to death. Would average individuals actually use markup to model their realities? Look at what they do with spreadsheets and the various cloud integration toolkits. This is an accessibility/ price/ performance thing.
>
> —Test subject

3.3.2. Organizational impacts

> Humanity’s legacy of stories and storytelling is the most precious we have. All wisdom is in our stories and songs. A story is how we construct our

> experiences. At the very simplest, it can be: ‘He/she was born, lived, died.’ Probably that is the template of our stories – a beginning, middle, and end. This structure is in our minds.
>
> —Doris Lessing, author

Markup has strategic organizational impacts. That's why global publishers use it.

Markup helps authors make sense of the past, present, and future. It makes it easier to organize information for efficient, increasingly-customized and personalized pipelines. This is human behavior we're trying to influence. We're not just dealing with data.

Figure 8. Authoring meaning through content, structure and style

```
Past meaning      \   Observe    / Future meaning
Established values \     O      /  Future values
Past practice       \    ||    /   Future behavior
Source origins      /    /\    \   Downstream k flows
Old logic          /   Orient   \  Re-contextualized logic
Event & prior K   / Decide & Act \ Learnings
```

Markup is good for top-down communications and alignment. If top-down alignment systems were sufficient, our existing single-source-based communication and publishing systems should be ensuring top-notch performance.

But operational realities in countless natural economies point to the need for fundamental changes in the rules of the symbolic/ market economy. That appears to be becoming increasingly difficult as the mechanisms that influence market behavior appear to be cutting themselves off from any communications that might challenge the current systems for monetizing human behavior.

Through countless decisions, systems evolve and are optimized around specific value propositions. This has happened to markup systems and associated technologies. The ISO:SGML platform, quite simply, establishes a benchmark standard for open systems. XML's refinements have brought countless new voices to the table, but author-authored markup has simply not gotten the same attention as single-sourcing and those value optimizations are reflected in the available technology alternatives through many small details that increase authoring effort in subtle, but impactful ways.

Bottom-up sense making and communications are becoming more important. Enabling individuals to customize markup in an organizational setting can be expected to not only increase reuse and collaboration, but also better-enable multi-perspective systems and decisions.

Formalizing individual value-optimizations as semantic values shares many traits with fundamentally-enabling technologies, like email. It opens up new channels of communication. How many organizations justified their initial investments in email with its impact reducing paper mail. That happened, but it didn't anticipate the completely-new models of communication that were enabled.

> The big experiment: Can bottom-up knowledge flows be established quickly-enough to build the new understandings and consensus around the logic needed to rapidly realign global supply chains around strategic physical and operational constraints.
>
> —Primary researcher

4. Document-level controls to capture and communicate meaning

> Semantic authoring is like car restoration on *Full Custom Garage.* The markup is the frame, and the CSS is the sheet metal. Pound at will. Make the data dance and drive change.
>
> —Test subject

prodoc provides two primary ways to influence the meaning of information: data structures and visualizations.

The basic process for making sense of new data?

1. Do existing structures work? If so, adapt, otherwise...
2. Add markup to give the data structure
3. Pick meaningful names
4. Add visual differentiation
5. Adjust the look and geometry to look for patterns
6. Rinse and repeat until the data becomes actionable
7. Add a switch or something to change the appearance in real time, if it helps

Sometimes the process starts by making the data look pretty before creating custom data structures. The ability to incrementally add structures and style rules to the base platform reduces the incremental cost of small projects. Successful experiments get moved from the laboratory document to prodoc.dtd.

In practice, document-level experimentation has been key to the evolution of prodoc. The full case study document is the actual file that started the development of the `<music/>` element.

Some of the design alternatives were driven by knowledge loss. The old markup didn't make sense. Create new markup that makes sense for current thinking. A few data structures only stabilized after multiple documents with partial solutions were created, sometimes across years. A couple of systems only went into production after the best features of 3 or 4 solutions from different perspectives were integrated.

4.1. Author-driven structural changes

> Fast, cheap, and out of control
>
> —Kevin Kelly

Most publishing systems use slow moving schema and stylesheets to ensure consistency of authoring. Semantic authoring is intended to be fast moving. Allowing ideas to be quickly captured and different approaches tested. Creating ecosystems of markup alternatives, driven by personal and small-group imperatives.

> Running off to a DTD file? Too far away, too slow, too impactful. I'm the authority here. Let me extend structures in the document.
>
> —Test subject

> Respect my authoritah
>
> —Eric Cartman

This example of enabling SGML Internal Declaration Subset extensions comes from h1.dtd, a training doctype. An updated oXml framework will be released on June 10th, 2023.

Elements are organized into four non-overlapping groups: `%divs;`, `%blocks;`, `%heads;`, and `%spans;`, so that they can be easily extended and combined, at will. This is the example of the definitions for `%divs;`

Figure 9. `%divs;` declarations includes the `%sa.divs;` extension mechanism for use in documents

```
<!-- __________________________________________________________________________
. . {
. . {         %divs; - infinitely-nesting wrapper elements
. -->
<!-- ..........................................................................
. . :                 %h1.divs;         . baseline hierarchical divisions
. -->
 <!ENTITY % h1.divs           " div|
                                h1|
                                hsa|
                                hsg|
                                prodoc" >
<!-- ..........................................................................
. . :                 %sa.divs;         . semantic authoring interface
. -->
 <!ENTITY % sa.divs           "" >
<!-- ..........................................................................
. . :                 %divs;            . (roll-up)
. -->
 <!ENTITY % divs              " %h1.divs;
                                %sa.divs;" >
```

Figure 10. Document header, where new elements are added through the `%sa.divs;` interface

```
<!DOCTYPE div PUBLIC "-//SG//DTD h1//EN" "../h1.dtd"[
<!ENTITY    h1                "h1" >
<!ENTITY  % sa.divs           "" ><!-- e.g., "| ename" -->
<!ENTITY  % sa.blocks         "" >
<!ENTITY  % sa.heads          "" >
<!ENTITY  % sa.spans          "" >
<!ENTITY  % sa.atl            "" >
<!ELEMENT   ename             (#PCDATA) >
<!ATTLIST   ename
              aname           CDATA                                  #IMPLIED >
]>
```

A common reason for adding spans to a document is to customize tables. `<tr/>` contains `%spans;`, and the set of `%spans;` includes `<td/>` and `<th/>`. Adding any element to the set of `%spans;` adds them to table rows.

4.2. Author-driven visual changes

> “Quickly changing the look and feel of text is WP's core value proposition. Pretty, but dumb. And it takes a lot of mousing around to set the properties on all those unrelated objects. Hurts my arm. Let me keep my keys on the keyboard, please. Let me apply style changes to branches and not leaves.”
>
> —Test subject

Since the semantic authoring platform was developed to displace WordPerfect, various markup-based style controls (control surfaces) were implemented.

Elements can function as visual control surfaces. `<p/>` adds whitespace. `<t/>` doesn't. `<frame/>` adds a border. `<block/>` doesn't. Mapping the oXmlRename Element function to Alt-N enables quickly changing how data looks by changing the element name. This frequently happens when tuning inline markup to adjust prominence.

Global attributes provide most of the visual control surfaces, overriding the CSS stylesheet defaults. Some of the simplest attributes are direct pass through. `@borders`, `@padding`, and `@style` accept standard CSS syntax. Many css properties were renamed to simplify authoring: `@bgcolor`, `@p.left`, `@p.right`, `@face` (typeface).

Some attributes add support for fixed values. `@scale`, for example, sets font-size using arbitrary values. `@sgscale` contains a standardized and generalized set of values based on the square root of the golden ratio. It started as a reference, for quickly bringing up a list of values, before being given an active role.

`@color` and `@bgcolor` are defined with a number of named colors that map to a color library. One of the color pallets is for modeling knowledge flows. Another attribute, `@kstyle`, applies background, foreground, and border color combinations based on the value of the `@k` attribute, which describes the element's role in knowledge flows.

A few controls are more aspirational than functional. `@lineheight` never seems to work. `@max-height` and the pagination settings are waiting for more powerful engines.

A variety of `@sh*` attributes show and hide control panels, table grids, id values, purple numbers, included content, and `<xi:include/>` controls. A couple more activate CPU-crushing focus and hover behaviors.

5. prodoc in practice

One day, the test subject was amused by the total absurdity of keeping project notes and analysis in WordPerfect (WP) while simultaneously writing XSLT specifications using an XML editor. The test subject rebuilt that specification doctype for a project and started modifying it to replace WP for all professional documentation. The customized authoring and publishing platform was named “prodoc.”

Technically, prodoc is an , centered around `prodoc.dtd`, and a supporting toolkit, with many of the tools based on Oxygen XML Editor and Author software (oXml).

And so started a grand experiment. Could someone who had provided training, support, and engineering to thousands of WP users transition?

The only “active” WP document is business cards that are encoded in tables with offset crop marks.

> Go back to WordPerfect? Are you crazy? Here, I can grab an element by pressing Alt-E. Even copying and pasting is easier. When I go to another editor, none of my compositional idioms are there. Yuck!
>
> —Test subject

> I don't know how I could move these documents back to a word processor. Even accepting the loss of structure, the algorithmic visual effects would take too many mouse clicks to be worth the effort, and probably require a separate graphics package.
>
> —Test subject

5.1. @class – Authored class styles

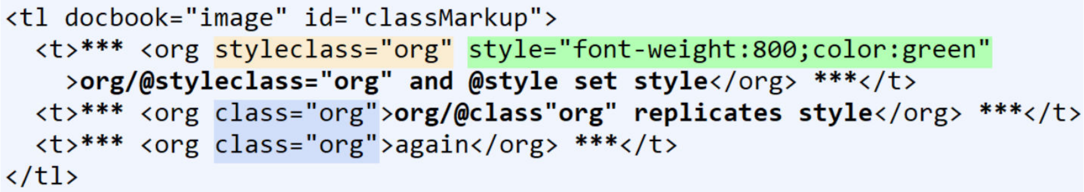
> “Look Ma. No stylesheet changes”
>
> —Test Subject

An early design principle was, “Hands off the @class attribute. That's for user control.” Recently, an approach was implemented that allows authors to define classes and associate specific CSS styles, all within a document.

The @style attribute maps to the oXml -oxy-style CSS property. This allows raw CSS to be passed to the CSS processor for the defined element.

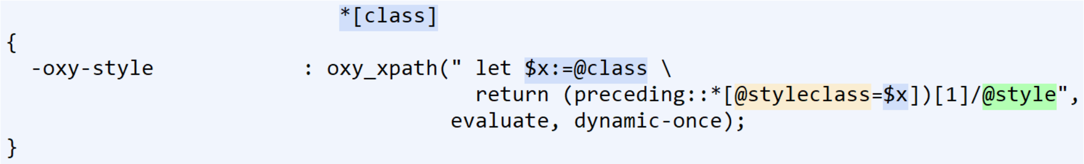
Adding a @styleclass attribute on an element makes the @style value available for reuse. A little XPath handles the indirection from @class, through the preceding @styleclass, to @style:

Figure 11. @class markup



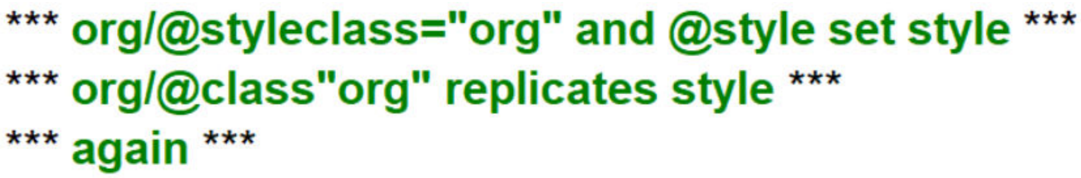
```
<tl docbook="image" id="classMarkup">
  <t>*** <org styleclass="org" style="font-weight:800;color:green"
    >org/@styleclass="org" and @style set style</org> ***</t>
  <t>*** <org class="org">org/@class"org" replicates style</org> ***</t>
  <t>*** <org class="org">again</org> ***</t>
</tl>
```

Figure 12. @class CSS



```
                              *[class]
{
  -oxy-style              : oxy_xpath(" let $x:=@class \
                                  return (preceding::*[@styleclass=$x])[1]/@style",
                                evaluate, dynamic-once);
}
```

Figure 13. @class rendering



*** org/@styleclass="org" and @style set style ***

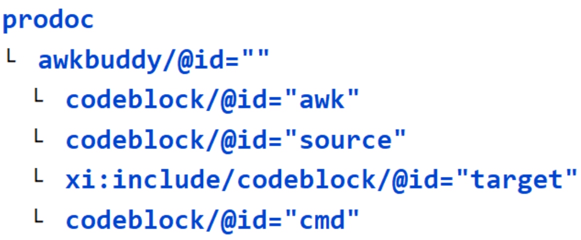
*** org/@class"org" replicates style ***

*** again ***

5.2. <awkbuddy/> – An interactive development environment block

One day, dreaming about the keyboard macro processors, I thought, “AWK! of course, AWK!” The result is <awkbuddy/>, which inverts most code/ documentation conventions by putting code fragments into a standard prodoc.

Figure 14. <awkbuddy/> codeblocks



```
prodoc
└ awkbuddy/@id=""
  └ codeblock/@id="awk"
  └ codeblock/@id="source"
  └ xi:include/codeblock/@id="target"
  └ codeblock/@id="cmd"
```

The code blocks can be arranged, as desired, to co-locate knowledge with decisions. I've tuned mine on eye movements, by wrapping the codeblocks in table structures. The `awkbuddy.cmd` script rips the prodoc into separate files and runs the pipeline. Factoring out the file-processing overhead made AWK a much more convenient tool. More little bots have been built. Much time saved. Much more complexity managed.

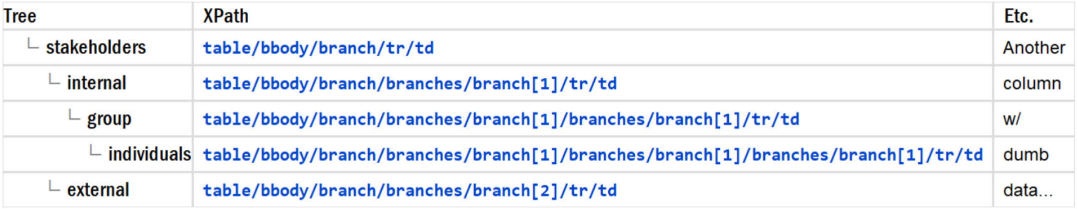
Perhaps more importantly, it improves knowledge retention and library accessibility. Simply having the code and a source fragment is usually sufficient to remember the project. On the other hand, if you want to write a novel to explain a few lines of AWK, the tools are at hand.

Custom IDs can be used to run multiple awkbuddies from the same prodoc and prevent collisions in the generated files. File-level collisions across separate awkbuddies (`*.abud`) never proved to be much of an issue. If the default target fragment has been overwritten, just rerun the transform.

5.3. `<bbody/>`,`<branches/>`,`<branch/>` – Hierarchical tables

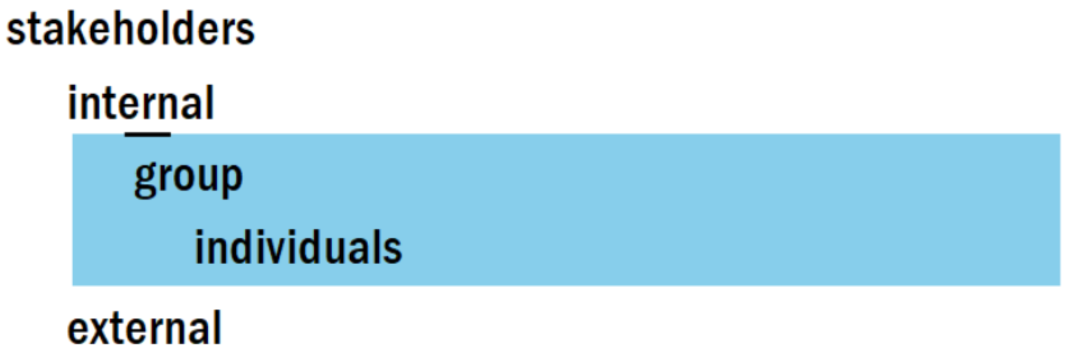
The first hierarchical table was developed to do stakeholder analysis. The generalized implementation uses a specialized table body element: `<bbody/>`. Changing `@display` switches between the full table and a simplified blocks view, which makes it easier to restructure the branches.

Figure 15. `table/@display="table"` (columns displayed for data processing)



Tree	XPath	Etc.
└ stakeholders	`table/bbody/branch/tr/td`	Another
└ internal	`table/bbody/branch/branches/branch[1]/tr/td`	column
└ group	`table/bbody/branch/branches/branch[1]/branches/branch[1]/tr/td`	w/
└ individuals	`table/bbody/branch/branches/branch[1]/branches/branch[1]/branches/branch[1]/tr/td`	dumb
└ external	`table/bbody/branch/branches/branch[2]/tr/td`	data...

Figure 16. `table/@display="block"` (columns hidden for tree processing)



stakeholders

internal

group

individuals

external

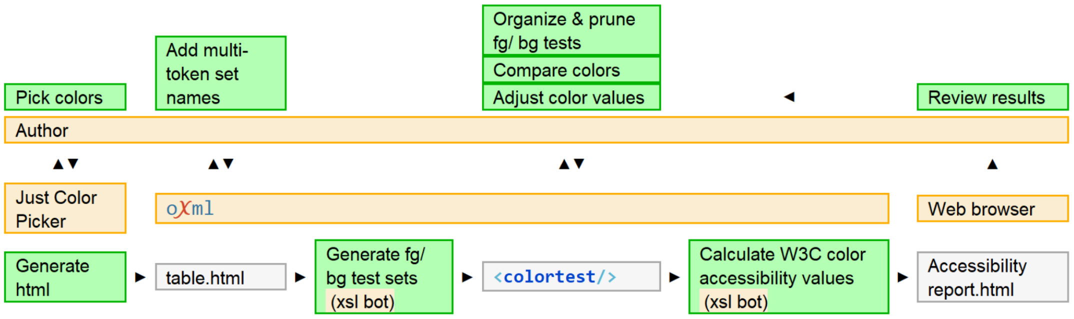
Figure 17. <bbody/> markup

```
<table shindent="show" hs="corner" display="block" shfocus="show">
  <bbody>
    <branch>
      <tr>
        <th id="th">stakeholders</th>
        <td><xpath>table/bbody/branch/tr/td</xpath></td>
      </tr>
      <branches>
        <branch>
          <tr>
            <th>internal</th>
            <td><xpath>table/bbody/branch/branches/branch[1]/tr/td</xpath></td>
          </tr>
          <branches>
            <branch>
              <tr>
                <th>group</th>
                <td><xpath>table/bbody/branch/branches/branch[1]/</xpath><xpath>branches/branch[1]/tr/td</xpath></td>
              </tr>
              <branches>
                <branch>
                  <tr>
                    <th>individuals</th>
                    <td><xpath>table/bbody/branch/branches/branch[1]/<xpath>branches/branch[1]/<xpath>branches/branch[1]/tr/td</xpath></xpath></xpath></td>
                  </tr>
                </branch>
              </branches>
            </branch>
          </branches>
        </branch>
        <branch>
          <tr>
            <th>external</th>
            <td><xpath>table/bbody/branch/branches/branch[2]/tr/td</xpath></td>
          </tr>
        </branch>
      </branches>
    </branch>
  </bbody>
</table>
```

5.4. <colortest/> – Automating accessible color negotiation pipelines

Challenge: Use the W3C-published formula for calculating the accessibility of color combinations. Over many years, the system was re-factored. Copying and pasting values into a small XML model was fine for small projects, but I had to seek client approvals on a pallet of 24 colors that were going to be used for strategic branding.

Figure 18. Color testing and tuning pipeline


Pick colors
Add multi-token set names
Organize & prune fg/ bg tests
Compare colors
Adjust color values
◄
Review results
Author
▲▼
▲▼
▲▼
▲
Just Color Picker
oXml
Web browser
Generate html
►
table.html
►
Generate fg/ bg test sets (xsl bot)
►
<colortest/>
►
Calculate W3C color accessibility values (xsl bot)
►
Accessibility report.html

The Just Color Picker tool generated HTML. A new transformation was created to map the HTML output to a new prodoc element, <colortest/>. The production accessibility calculator was updated to handle the new schema and missing functionality was added.

Implementation had a mix of performance objectives: large swatches to make small comparisons, built in black, gray, and white samples, ability to instantly see color changes based on editable hex values, demonstrate single-sourcing concepts to client, save me an armload of effort

5.5. <h/> – Depth-based headings because big headings are ugly

HTML uses numbered headings, prodoc, on the other hand, uses nested divs with a single, context-sensitive <h/> element

The following XPath scales headings based on the maximum depth of the document. 1.272 is the square root of the golden ratio [https://www.modularscale.com/?1&em&1.272].

A recent document needed a gentler scaling factor. The cube root of the golden ratio (1.174 [https://www.modularscale.com/?1&em&1.174]) was used, instead.

Figure 19. CSS to scale headings based on document depth

```
h {
font-size : oxy_xpath("concat(
            math:pow(1.272,((max(//h/count(ancestor::*[@display='div'])) - count(ancestor::*[@display='div']) + 1 ))),
            'em')", evaluate,dynamic-once) ;
}
```

5.6. <kfam/> – An element and generalized design language to make sense of knowledge flows from multiple perspectives

> The kfam ontology includes a lot of concepts, but you only really need to focus on three to make sense of knowledge flows: agent, artifact, and behavior.
>
> —Test subject

The knowledge flow analysis and modeling language — kfam — represents the most extreme example of the fractal nature of this approach to computer-assisted sense making.

The base language, markup, and visualizations associated with kfam have evolved over 30 years of discussions around organizational performance. New concepts and ideas around those concepts interplay with new ways to organize and control the appearance of kfam data. The result is a design language, where kfam concepts are used to name elements, attributes, colors, etc.

One view of sense making is answering the question, “How to make sense of this data? How to organize it without incessantly banging on the keyboard?”. The meaning of kfam from this perspective centers on ergonomics, the behavior of eyes and hands. Operationally, that's tuning the markup, interfaces, and bots to organize data into actionable knowledge.

When multiple datasets need the same treatment, shifting from manual to automated processes makes sense. This type of computer-assisted sense making —making sense of new data sources — would often be considered secondary deliverables, infrastructure that enables behavior, lowers costs, and improves quality.

When kfam is applied to primary activities, the questions shift to, “How do I make sense of this situation. What are we trying to accomplish? Who knows what? What doesn't know enough?”

This type of sense making focuses on making sense of a behavioral domain, the performance objectives, and the specific knowledge requirements that enable intelligent behavior.

<kfam/> element is a customized table that evolved to help the Test subject make sense of:

- ◇ The ways that knowledge flows through organizations and is acted upon, operationally, planned, and implemented
- ◇ The various bits and pieces of XML system architectures At its core, kfam is a language for dealing with the fractal nature of language and behavior.
- ◇ Competing, multidimensional value optimizations. The variety of performance objectives and the specific knowledge requirements that enable intelligent behavior within a span of control.

When engineering markup systems, analysis of knowledge flows has impacts on architectural decisions, fine-grained markup decisions, usability, and occasionally how to deal with organizational dynamics.

By being closely associated with primary work products and being rather abstract, the meaning of kfam concepts has been highly-dynamic, making formalization an iterative process. Of all of the systems built on the prodoc platform, kfam has had the most iterations and has the most moving parts. It is the most expansive example of semantic authoring being used to create markup based on a person's conceptual models.

> I started working with markup at the same time that Bo introduced me to knowledge management and Joe, values-based decision making. I couldn't untangle those conceptual frameworks with a Lampson crane.
>
> —Test subject

5.6.1. kfam conceptual language

kfam's organized around three core concepts: agents, artifacts, and behaviors. Those three concepts are sufficient to describe a knowledge flow.

Agentsact, make decisions, and exhibit behavior. The set of agents comprise individuals, organizations, systems, and automated agents (bots).

Artifacts are physical or conceptual objects.

Knowledge can also be characterized through such distinctions as

- Knowledge artifacts (ka) and meta-knowledge artifacts (mka)
- Data, information, knowledge
- Explicit, implicit, and tacit forms

Behavior comprises actions and decisions. Knowledge enables behavior. Co-location of knowledge and decisions generally improves performance. Behavior goes by many different names in different contexts.

Policy is what you do. Policy is behavior. Written policies are only guidance. The pages don't act. Ideally, they are semantic triggers, triggering the intended behaviors by the acting agents.

Meaning is behavior. Words have no meaning without action. Knowledge flows and lifecycles comprise such behaviors as knowledge creation, retention, transfer, use, and destruction.

Values are disassociated and abstracted knowledge systems that synthesize pattern recognition and evaluation to quickly apply established logic to operational details with minimal thought. They represent distilled and automated behaviors. Unscripted life requires active management systems; automate the operational details. Novelty focuses active attention, thinking, and — if there isn't too much fear — creativity.

While agent, artifact, and behavior are enough to describe a knowledge flow, additional details help with detailed analysis and engineering. The following diagram relates the kfam “data ∫ information ∫ knowledge ∈ behavior” model with Sowa's concepts and OODA loops, the process that fighter pilots use to Observe, Orient, Decide, and Act.

Figure 20. Knowledge flows associated with knowledge enabling behavior

5.6.2. kfam markup language

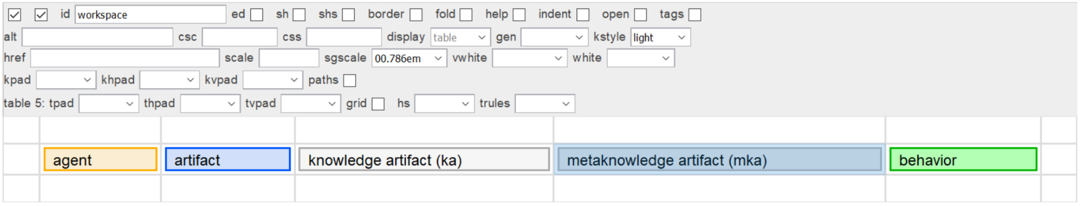
In markup systems, elements are first-class objects. Early experiments with extending and customizing semantics using attributes proved unworkable due to accessibility issues.

A couple of other design principles: The egg carton principle. How many eggs would you buy if you had to create your own egg cartons, on the spot? Be generous when you build containers. Redundancy, many ways to say the same thing, doesn't hurt.

Formalizing kfam starts by creating `%spans;` based on the concepts, e.g., `<agent/>`, `<artifact/>`, `<behavior/>`. Likewise for attributes. `%semantic-atl;` includes `@agent`, `@artifact`, and `@behavior`.

Most modeling is done in a specialized table called `<kfam/>`, which uses a restricted element set and has a fairly extensive set of controls:

Figure 21. `<kfam/>` workspace



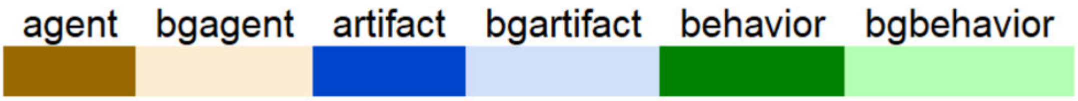
☑ ☑ id workspace ed □ sh □ shs □ border □ fold □ help □ indent □ open □ tags □
alt csc css display table gen kstyle light
href scale sgscale 00.786em vwhite white
kpad khpad kvpad paths □
table 5: tpad thpad tvpad grid □ hs trules

	agent	artifact	knowledge artifact (ka)	metaknowledge artifact (mka)	behavior	

5.6.3. kfam visual language

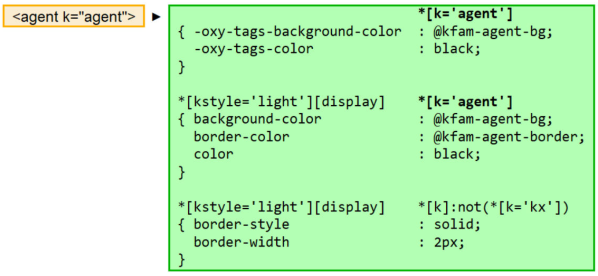
The kfam visual language starts with colors. Named colors are defined with HSL values and variables in LESS stylesheets. The names are defined for use as attribute values in prodoc.dtd so that they can be applied to `@color` and `@bgcolor` using picklists. One of the color pallets is for modeling knowledge flows.

Figure 22. Named foreground and background colors



Another attribute, `@kstyle`, applies `<kfam/>`visualizations to any element. `@kstyle` accepts the values `(bgcolor| color| custom| dark| full| gray| kflow| light| line| none)`. It applies background, foreground, and border color combinations based on the value of the `@k` attribute, which describes the element's role in knowledge flows. prodoc.dtd sets default `@k` attribute values for most spans, which can be overwritten to lie to the author, e.g., `agent/@k="behavior"`, would create a secret `<agent/>` camouflaged to look like a `<behavior/>`.

Figure 23. `@k` → `@kstyle` CSS style mappings

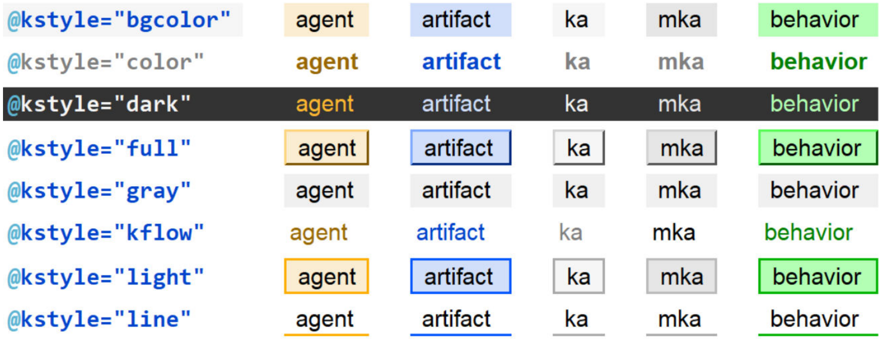


```
                                  *[k='agent']
{ -oxy-tags-background-color    : @kfam-agent-bg;
  -oxy-tags-color               : black;
}

*[kstyle='light'][display]      *[k='agent']
{ background-color              : @kfam-agent-bg;
  border-color                  : @kfam-agent-border;
  color                         : black;
}

*[kstyle='light'][display]      *[k]:not(*[k='kx'])
{ border-style                  : solid;
  border-width                  : 2px;
}
```

Figure 24. `@kstyle` variations applied to a table



@kstyle="bgcolor"	agent	artifact	ka	mka	behavior
@kstyle="color"	agent	artifact	ka	mka	behavior
@kstyle="dark"	agent	artifact	ka	mka	behavior
@kstyle="full"	agent	artifact	ka	mka	behavior
@kstyle="gray"	agent	artifact	ka	mka	behavior
@kstyle="kflow"	agent	artifact	ka	mka	behavior
@kstyle="light"	agent	artifact	ka	mka	behavior
@kstyle="line"	agent	artifact	ka	mka	behavior

5.6.4. kfam modeling

kfam started as a natural language for discussing project activities and how knowledge enables behavior. Some of that ambiguity has remained as the language has evolved into an analytical system, modeling language, and engineering framework. The most complete, published definition of kfam can be found in *Knowledge Flow Modeling and Analysis with Focus on Enabling Actions and Decisions within the Business Process* [https://sagebrushgroup.com/archive/201009_KFlowAnalysisModel_Paper.pdf].

kfam was first used as a modeling language when markup program managers were being blamed for expected IT schedule slippages. Finding the solution was pretty easy.

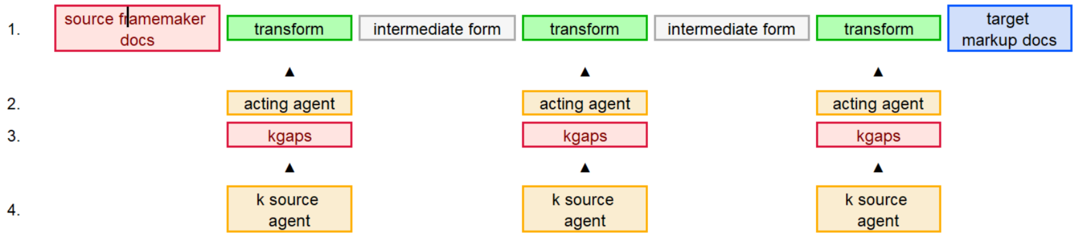
1. Map out the processing steps, defining each of the transforms and intermediate data forms

2. Identify the responsible agents. Who was doing the work? A person? A bot? Some combination?

3. Any knowledge gaps or other barriers to progress?

4. Where does any missing information need to come from?

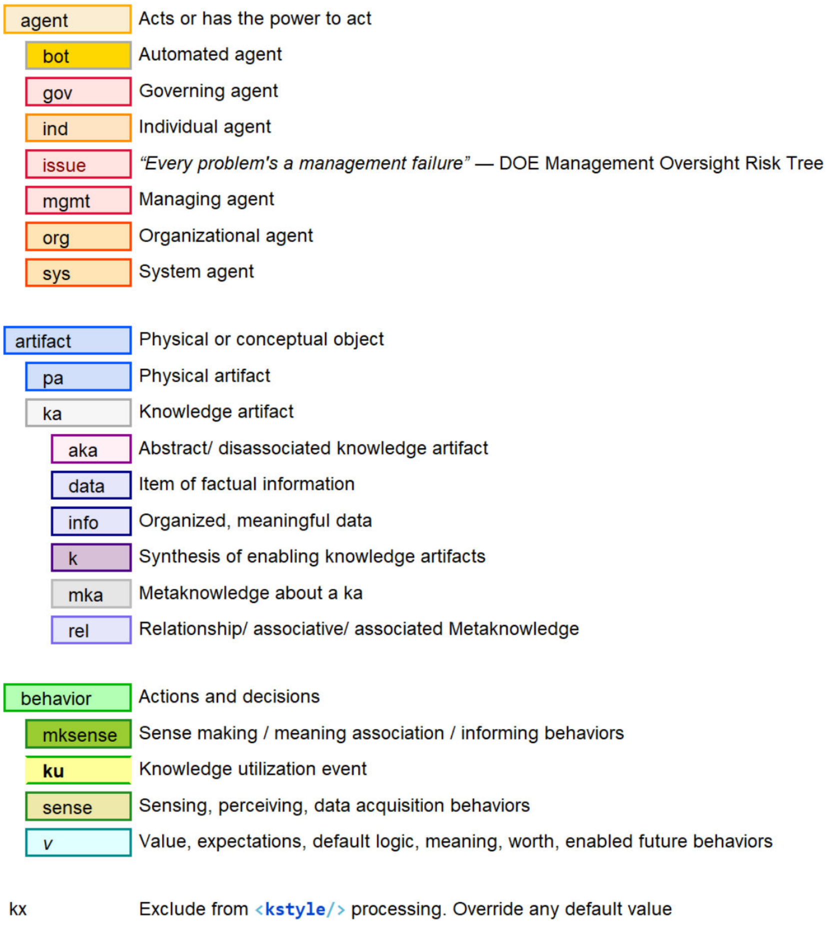
Figure 25. Knowledge gap analysis



It turned out that all of the knowledge gaps were pending IT decisions. The problem went away. We also made a couple of refinements to the startup process.

Here's a more-complete set of kfam elements: Any element can join a class by setting the @k attribute to the listed value.

Figure 26. @k values and associated colors

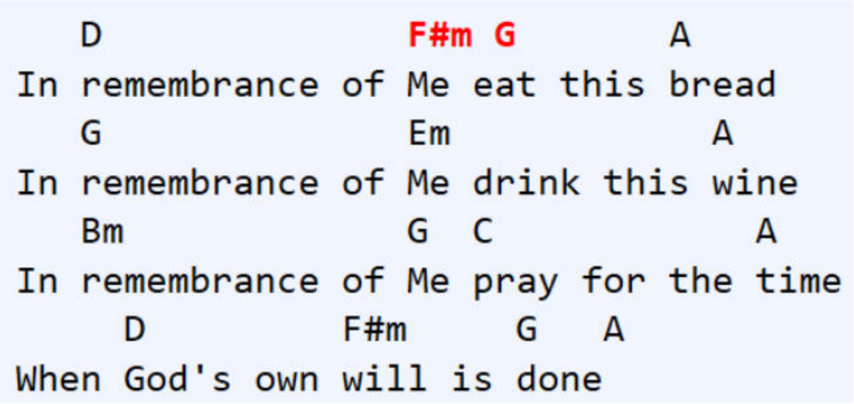


agent	Acts or has the power to act
bot	Automated agent
gov	Governing agent
ind	Individual agent
issue	*"Every problem's a management failure"* — DOE Management Oversight Risk Tree
mgmt	Managing agent
org	Organizational agent
sys	System agent
artifact	Physical or conceptual object
pa	Physical artifact
ka	Knowledge artifact
aka	Abstract/ disassociated knowledge artifact
data	Item of factual information
info	Organized, meaningful data
k	Synthesis of enabling knowledge artifacts
mka	Metaknowledge about a ka
rel	Relationship/ associative/ associated Metaknowledge
behavior	Actions and decisions
mksense	Sense making / meaning association / informing behaviors
ku	Knowledge utilization event
sense	Sensing, perceiving, data acquisition behaviors
v	Value, expectations, default logic, meaning, worth, enabled future behaviors
kx	Exclude from <kstyle/> processing. Override any default value

5.7. <music/> – Rationalizing chord/ lyric pairings

When chord symbols lose alignment with the lyrics, they lose their meaning. With WP, the pairings regularly fall out of alignment for countless reasons:

Figure 27. Lyric-chord charts with misaligned text (it gets much worse, especially with proportional fonts)



```
   D               F#m G       A
In remembrance of Me eat this bread
   G               Em            A
In remembrance of Me drink this wine
   Bm              G  C            A
In remembrance of Me pray for the time
     D          F#m     G   A
When God's own will is done
```

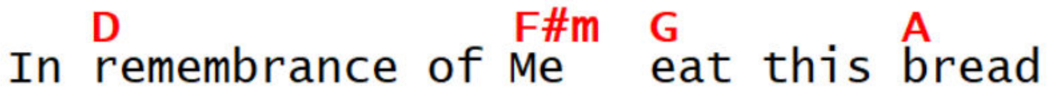
Markup was used to make the chord <c/> and lyric <l/> relationships more explicit.

Figure 28. `<lyric/>` lines, with lyric fragments `<l/>`, and chord `<c/>`symbols next to each other so they don't get lost

```
<music>
  <lyric>
    <l>In </l>
    <c>D</c>
    <l>remembrance of </l>
    <c>F#m</c>
    <l>Me </l>
    <c>G</c>
    <l>eat this </l>
    <c>A</c>
    <l>bread</l>
  </lyric>
</music>
```

XPath was used to manage horizontal and vertical offsets.

Figure 29. CSS rendering with fully-automated offsets



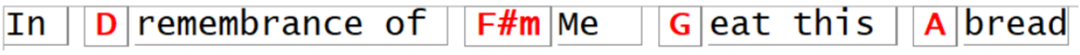
```
          D                          F#m  G                A
In remembrance of Me    eat this bread
```

All the CSS gymnastics cause cursor-positioning problems. Adding an edit mode solved the problem:

Figure 30. `music[@view="edit"]` markup

```
<music view="edit">
  <lyric>
    <l>In </l><c>D</c><l>remembrance of </l><c>F#m</c><l>Me&nbsp;&nbsp;</l>
    <c>G</c><l>eat this </l><c>A</c><l>bread</l>
  </lyric>
</music>
```

Figure 31. `music[@view="edit"]` rendering

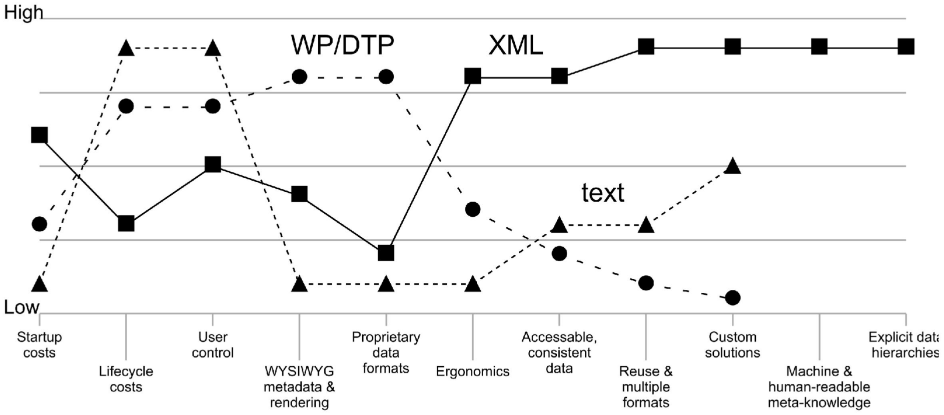


In | D | remembrance of | F#m | Me | G | eat this | A | bread

5.8. `<vcanvas/>` – Visualizing and comparing sets of value optimizations

Value canvases come from the book *Blue Ocean Strategy*, They provide a good way to visualize and compare multiple sets of value propositions. The following SVG rendering was generated from data entered into a `<vcanvas/>` table.

Figure 32. Rendering of <vcanvas/> datasets



5.9. WordNet and SUMO integrations – Associating markup with dictionaries & formal logic

There are different ways to organize bottom-up markup. A technical solution is namespaces, but that just keeps the chaos from bugging the bots.

> Namespaces! A linguistic solution without behavioral context (e.g., requirements). So many different implementation models. So much fun.
>
> —Anonymous source

Another approach is negotiations. As individual markup is shared and encounters similar/overlapping markup, conversations are used to generalize the concepts and markup. Different concepts get different names. Individual markup feeds departmental standards. Semantic generalization flows up through layered DTDs, matching the organizational hierarchy. This represents an evolutionary approach to bottom-up data modeling and system design.

Coming at the question from the opposite direction, individual markup could be anchored to a separate, semantic authority. At the first Ontolog face-to-face meeting & workshop in 2003, Adam Pease introduced SUMO, the Suggested Upper Merged Ontology to the group. The aftermath is described in the forward to In Adam's book *Ontology*:

> The result of [Adam's] behavior was quite infectious. Adam, Kurt Conrad, and I ended up in a late night sushi restaurant somewhere near Menlo Park, CA, discussing how to map SUMO concepts to Mandarin, Japanese, and Cantonese, how WordNet can reference SUMO, and why First-Order Logic (FOL) constraints are generally a cool concept to have in advanced computer systems. An upper-level ontology, such as SUMO, is a common, shared conceptualization of a domain.
>
> —Duane Nickull

5.9.1. Approach

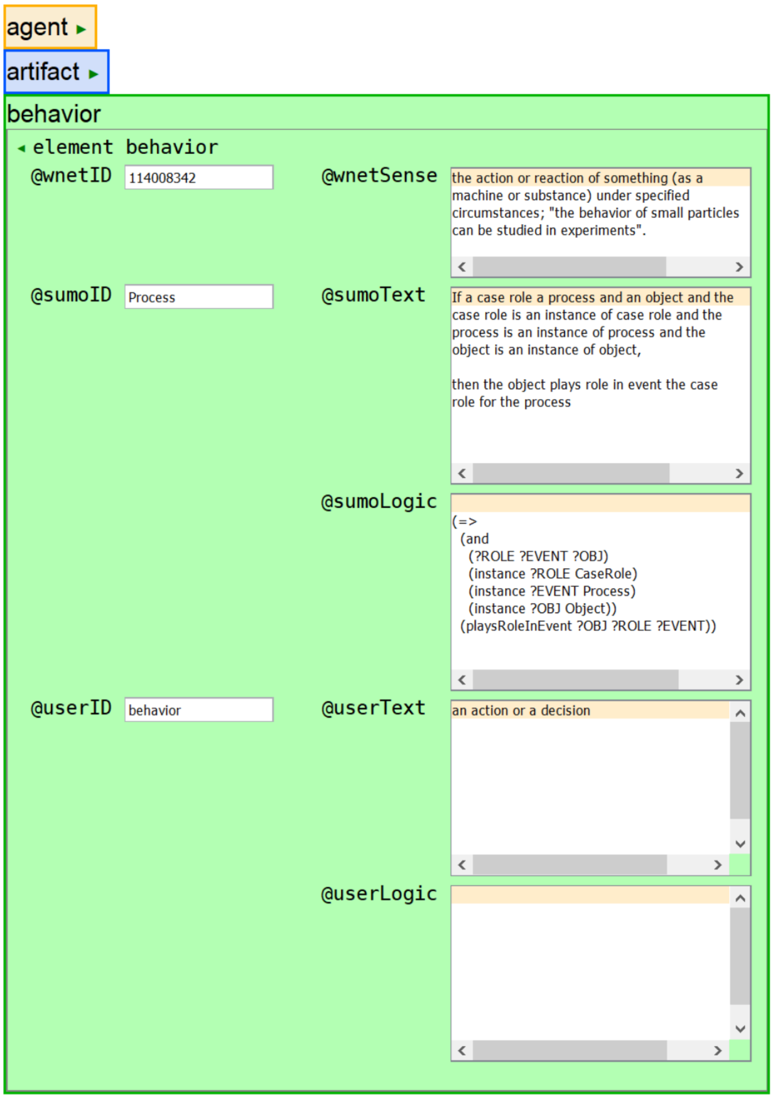
The draft implementation uses prodoc's document-level extension mechanism to add global attributes a test document:

Figure 33. Semantic formalization attribute list

```
<!-- ren to fsem.atl when generalized -->
<!ENTITY  % sa.atl
"
              shsem           (show| hidden)   #IMPLIED
              sumoID          CDATA            #IMPLIED
              sumoLogic       CDATA            #IMPLIED
              sumoText        CDATA            #IMPLIED
              userID          CDATA            #IMPLIED
              userLogic       CDATA            #IMPLIED
              userText        CDATA            #IMPLIED
              wnetID          CDATA            #IMPLIED
              wnetSense       CDATA            #IMPLIED
" >
```

The approach was tested with the concepts `<agent/>`, `<artifact/>`, and `<behavior/>`. The `@shsem` attribute controls the expansion and collapsing of the associated form control that provides access to the WordNet and SUMO attributes set.

Figure 34. Elements mapped to WordNet definitions and SUMO logic



@wnetID 114008342

@wnetSense the action or reaction of something (as a machine or substance) under specified circumstances; "the behavior of small particles can be studied in experiments".

@sumoID Process

@sumoText If a case role a process and an object and the case role is an instance of case role and the process is an instance of process and the object is an instance of object,

then the object plays role in event the case role for the process

@sumoLogic

```
(=>
 (and
  (?ROLE ?EVENT ?OBJ)
  (instance ?ROLE CaseRole)
  (instance ?EVENT Process)
  (instance ?OBJ Object))
 (playsRoleInEvent ?OBJ ?ROLE ?EVENT))
```

@userID behavior

@userText an action or a decision

@userLogic

5.9.2. Findings & next steps

> Humans are not ideally set up to understand logic; they are ideally set up to understand stories.
>
> —Roger C. Schank, cognitive scientist

An obvious next step is to connect the form control to a SUMO repository and add a lookup/ query interface. The level of effort appears reasonable.

More thought needs to go into a more challenging issue, authoring fully-formalized semantics. Spelling KIF is one thing. Spelling out formal logic in KIF is another.

Semantic markup can simplify knowledge transfer by making implicit organizing concepts more explicit. This is especially useful during learning, before a new conceptual framework's language and logic have been fully-internalized.

Expressing KIF concepts more explicitly through markup is expected to help it make sense in more behavioral contexts:

◇ For individuals, faster comprehension and productivity, especially when style variations can be applied to help differentiate concepts

◇ For the bots, more tool options. XSLT and XPath, by themselves, dramatically expand the software development options.

In his *Standard Upper Ontology Knowledge Interchange Format* document, Adam describes SUO-KIF using BNF syntax:

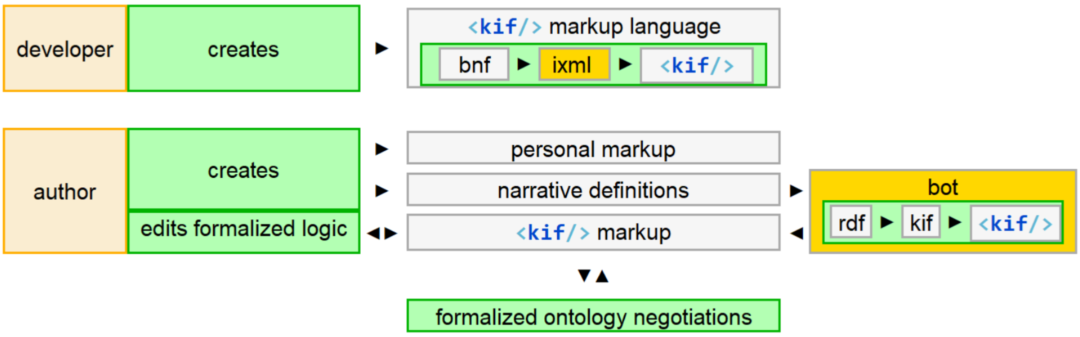
Figure 35. SUO-KIF definition

```
upper ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z
lower ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
special ::= ! | $ | % | & | * | + | - | . | / | < | = | > | ? |
@ | _ | ~ |
white ::= space | tab | return | linefeed | page
initialchar ::= upper | lower
wordchar ::= upper | lower | digit | - | _ | special
character ::= upper | lower | digit | special | white
word ::= initialchar wordchar*
string ::= "character*"
variable ::= ?word | @word
number ::= [-] digit+ [. digit+] [exponent]
exponent ::= e [-] digit+
term ::= variable | word | string | funterm | number | sentence
relword ::= initialchar wordchar*
funword ::= initialchar wordchar*
funterm ::= (funword term+) | (funword sentence+)
sentence ::= word | equation | inequality |
relsent | logsent | quantsent
equation ::= (= term term)
relsent ::= (relword term+) | (relword sentence+)
logsent ::= (not sentence) |
(and sentence+) |
(or sentence+) |
(=> sentence sentence) |
(<=> sentence sentence)
quantsent ::= (forall (variable+) sentence) |
(exists (variable+) sentence)
```

ixml was used to transform KIF to <kif/>. The most significant finding was that the definition appears to include a couple of unnamed layers that deserve clarification.

Integrations with other automated systems have been explored. OntoInsights focuses on the value propositions associated with storytelling and deep narrative analysis. It generates RDF fragments that can be translated into KIF.

Figure 36. From semi-formalized markup languages to fully-formalized ontologies



> Every time I look in one of those fully-formalized thingies, it never agrees with the way I think.

—Test subject

6. Bottom-up negotiations to define shared meanings

> Policy is what you do, not what you say or write.
>
> —Paul Strassmann, The Politics of Information Management

A few more mantras from Strassmann: “Information management is primarily a question of politics and secondarily one of technology.”“Operational decisions take care of today. Management decisions protect the future.”“Without privacy, there is no freedom.”

Negotiations around markup are rarely zero-sum. The ability to handle optional structures usually provides sufficient flexibility to accommodate different stakeholder interests. Even so, as the complexity of the project increases, integrating the multiple perspectives becomes more challenging.

Just as bottom-up markup starts with defining semantic values, a parallel, bottom-up approach to decision making starts with conversations to define common human values.

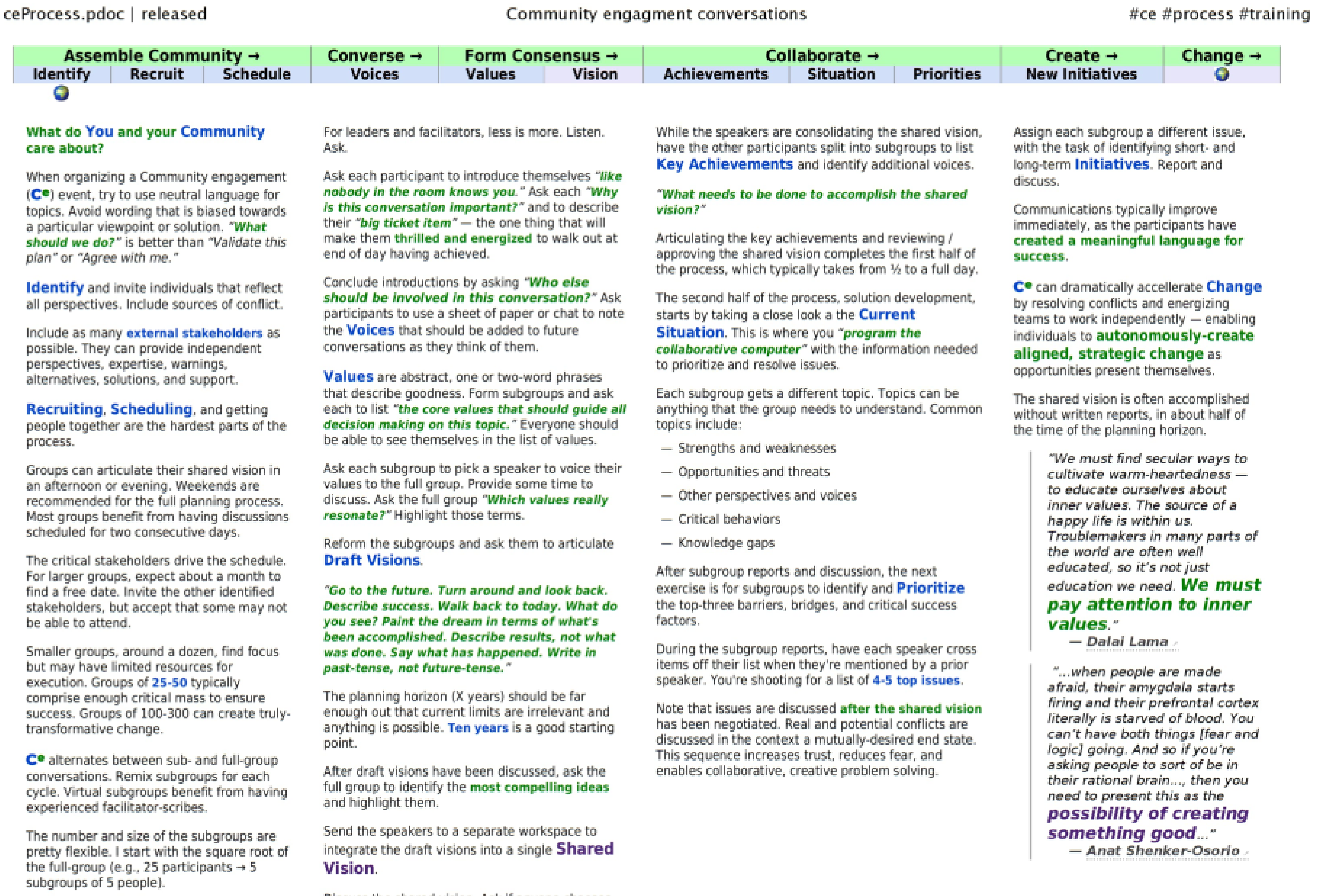
As the discussion topics progress, the group creates a shared vision for change and looks at the current conflicts from a position of safety. Issues are prioritized, and initiatives defined — quickly and without conflict.

More importantly, the process is energizing. Individuals see their values in the shared vision, and the new initiatives as a way to accomplish personal objectives.

From an ontological perspective, the identification, articulation, and prioritization of abstract values, starts a formalization process. The corresponding conversations create shared meanings to support the new language.

The following training aide describes the process in more detail:

Figure 37. Community engagement process



ceProcess.pdoc | released

Community engagment conversations

#ce #process #training

Assemble Community →			Converse →	Form Consensus →		Collaborate →			Create →	Change →
Identify	Recruit	Schedule	Voices	Values	Vision	Achievements	Situation	Priorities	New Initiatives	

What do You and your Community care about?

When organizing a Community engagement (**C•**) event, try to use neutral language for topics. Avoid wording that is biased towards a particular viewpoint or solution. ***"What should we do?"*** is better than *"Validate this plan"* or *"Agree with me."*

Identify and invite individuals that reflect all perspectives. Include sources of conflict.

Include as many **external stakeholders** as possible. They can provide independent perspectives, expertise, warnings, alternatives, solutions, and support.

Recruiting, **Scheduling**, and getting people together are the hardest parts of the process.

Groups can articulate their shared vision in an afternoon or evening. Weekends are recommended for the full planning process. Most groups benefit from having discussions scheduled for two consecutive days.

The critical stakeholders drive the schedule. For larger groups, expect about a month to find a free date. Invite the other identified stakeholders, but accept that some may not be able to attend.

Smaller groups, around a dozen, find focus but may have limited resources for execution. Groups of **25-50** typically comprise enough critical mass to ensure success. Groups of 100-300 can create truly-transformative change.

C• alternates between sub- and full-group conversations. Remix subgroups for each cycle. Virtual subgroups benefit from having experienced facilitator-scribes.

The number and size of the subgroups are pretty flexible. I start with the square root of the full-group (e.g., 25 participants → 5 subgroups of 5 people).

For leaders and facilitators, less is more. Listen. Ask.

Ask each participant to introduce themselves ***"like nobody in the room knows you."*** Ask each ***"Why is this conversation important?"*** and to describe their ***"big ticket item"*** — the one thing that will make them **thrilled and energized** to walk out at end of day having achieved.

Conclude introductions by asking ***"Who else should be involved in this conversation?"*** Ask participants to use a sheet of paper or chat to note the **Voices** that should be added to future conversations as they think of them.

Values are abstract, one or two-word phrases that describe goodness. Form subgroups and ask each to list ***"the core values that should guide all decision making on this topic."*** Everyone should be able to see themselves in the list of values.

Ask each subgroup to pick a speaker to voice their values to the full group. Provide some time to discuss. Ask the full group ***"Which values really resonate?"*** Highlight those terms.

Reform the subgroups and ask them to articulate **Draft Visions**.

"Go to the future. Turn around and look back. Describe success. Walk back to today. What do you see? Paint the dream in terms of what's been accomplished. Describe results, not what was done. Say what has happened. Write in past-tense, not future-tense."

The planning horizon (X years) should be far enough out that current limits are irrelevant and anything is possible. **Ten years** is a good starting point.

After draft visions have been discussed, ask the full group to identify the **most compelling ideas** and highlight them.

Send the speakers to a separate workspace to integrate the draft visions into a single **Shared Vision**.

While the speakers are consolidating the shared vision, have the other participants split into subgroups to list **Key Achievements** and identify additional voices.

"What needs to be done to accomplish the shared vision?"

Articulating the key achievements and reviewing / approving the shared vision completes the first half of the process, which typically takes from ½ to a full day.

The second half of the process, solution development, starts by taking a close look a the **Current Situation**. This is where you ***"program the collaborative computer"*** with the information needed to prioritize and resolve issues.

Each subgroup gets a different topic. Topics can be anything that the group needs to understand. Common topics include:

- Strengths and weaknesses
- Opportunities and threats
- Other perspectives and voices
- Critical behaviors
- Knowledge gaps

After subgroup reports and discussion, the next exercise is for subgroups to identify and **Prioritize** the top-three barriers, bridges, and critical success factors.

During the subgroup reports, have each speaker cross items off their list when they're mentioned by a prior speaker. You're shooting for a list of **4-5 top issues**.

Note that issues are discussed **after the shared vision** has been negotiated. Real and potential conflicts are discussed in the context a mutually-desired end state. This sequence increases trust, reduces fear, and enables collaborative, creative problem solving.

Assign each subgroup a different issue, with the task of identifying short- and long-term **Initiatives**. Report and discuss.

Communications typically improve immediately, as the participants have **created a meaningful language for success**.

C• can dramatically accellerate **Change** by resolving conflicts and energizing teams to work independently — enabling individuals to **autonomously-create aligned, strategic change** as opportunities present themselves.

The shared vision is often accomplished without written reports, in about half of the time of the planning horizon.

> *"We must find secular ways to cultivate warm-heartedness — to educate ourselves about inner values. The source of a happy life is within us. Troublemakers in many parts of the world are often well educated, so it's not just education we need.* ***We must pay attention to inner values.****"*
>
> — *Dalai Lama*

> *"...when people are made afraid, their amygdala starts firing and their prefrontal cortex literally is starved of blood. You can't have both things [fear and logic] going. And so if you're asking people to sort of be in their rational brain..., then you need to present this as the* ***possibility of creating something good****..."*
>
> — *Anat Shenker-Osorio*

7. h1.dtd – Build your own prodoc

`h1.dtd` is based on an HTML subset. It's scheduled for release on June 10th.

7.1. h1.dtd summary

```
<!-- ==========================================================================
                                                              dawkIns ....
              dawk Summary

    SPath:    D:\kc\xt\h1\
    SFile:    h1.dtd
    Gen:      2023-05-30 22:12
    GenBy:    D:\kc\xt\bin\dawk.awk
    Target:   D:\kc\xt\bin\scratch\dawk_tgt.txt
    fileType: xml
    chopCol:  79

    Module:   h1.dtd
              HTML 1 Document Type Declarations

    Public Identifier:

              -//SG//DTD h1//EN

    Date:     2021.05.29

    Version:  Released for testing

    Contents:

   175    Declaration constants (parameter entity declarations)
   178
   179      Extensible element constants
   182
   183        %divs; - infinitely-nesting wrapper elements
   186                %h1.divs;           baseline hierarchical divisions
   194                %sa.divs;           semantic authoring interface
   198                %divs;              (roll-up)
   203
   204        %heads; - top of div (metadata) elements
   207                %h1.heads;          baseline heading containers
   211                %sa.heads;          semantic authoring interface
   215                %heads;             (roll-up)
   220
   221        %blocks; - paragraph-level elements
   224                %h1.blocks;         baseline blocks (email benchmark)
   239                %sa.blocks;         semantic authoring interface
   243                %blocks;            (roll-up)
   248
   249        %spans; - inline elements
   252                %h1.spans;          baseline spans
   280                %sa.spans;          semantic authoring interface
   284                %spans;             (roll-up)
   289
   290      Attribute constants
   293
   294        Semantic attribute entities
   297                @alt                alternate (text for image)
   301                @agent              acts or has the power to act
   305                @artifact           physical or conceptual object
```

```
309                @author             agent responsible for writing text
313                @behavior           an agent's actions and decisions
317                @class              css classifications
321                @content            add @name for custom "attribute"
325                @created            when created
329                @href               hypetext reference
333                @id                 unique identifier
337                @idref              reference to a single identifier
341                @modified           when modified
345                @name               add @content for matching value
349                @src                media source
353                @status             relative ranking
357                @tags               #words that classify or categorize
361                @type               nature or genre
365                %sem.atl;           (roll-up)
385
386        Enumerated attribute value sets for style attributes
389                %avs.align;         horizontal text alignment values
394                %avs.display        display property values
402                %avs.open.closed;   details open/closed switch values
406                %avs.valign;        vertical alignment values
412                %avs.vwhite         vertical whitespace
417                %avs.whitespace     text whitespace
422
423        Style attribute entities
426                @align              horizontal text alignment
430                @bgcolor            background color
434                @border             (width style color)
438                @color              text color
442                @colspan            column span
446                @display            default display geometry is inline
450                %a.display.def;     defined default (e.g., "block")
454                @height             vertical size
458                @margin             whitespace ouside border
462                @open               open/close display & displayblock
466                @padding            whitespace inside border
470                @rowspan            row span
474                @scale              scales font size
478                @style              inline css
482                @valign             vertical alignment
486                @white              text whitespace
490                @width              horizontal size
494                @xspace.pre         xml:space="preserve"
499                %style.atl;         (roll-up)
517
518        %gatl; - extensible global attribute set
531                %sa.atl;            semantic authoring interface
535                %gatl;               (roll-up)
541
542      Element content model constants
545                %div.model;         generalized content model for divs
549                %fig.model;         content model for figure element
553                %kitchen.sink;      merged content models
557                %mixed;             parsable character data and %spans;
561                %text;              parsable character data content model
565                %undefined;         text
569
570    Element and attlist declarations
573
574      %h1.divs; declarations
577                div                 nested, hierarchical division
583                h1                  H1 wrapper
```

```
   589                hsa                 HTML for semantic authoring wrapper
   595                hsg                 HTML for semantic generalization wrap
   601                prodoc              prodoc wrapper
   607
   608      %h1.heads; declarations
   611                h                   heading
   617
   618      %h1.blocks; declarations
   621                block               generic block of text
   627                codeblock           block of source code
   634                figure              wrapper bordered content
   639                hr                  horizontal rule, thematic break
   644                ol                  ordered list wrapper
   649                p                   paragraph
   654                qblock              block formatting for quoted text
   659                t                   text
   665                table               tablular, matrix data
   670                  tr                table row
   675                tl                  text list
   680                ul                  unordered list
   685
   686      %h1.spans; declarations
   689                a                   hyperlink anchor
   694                agent               one who acts, that which acts
   699                artifact            physical or conceptual object
   704                b                   bold text
   709                behavior            action, decision, patterns
   714                big                 big text
   719                br                  line break
   724                details             @open="closed" hides all but summary
   729                i                   italic text
   734                img                 inline image
   742                meta                meta-data
   747                name                designates and references concept
   752                nowrap              non-breaking text
   757                q                   quotation
   762                s                   strikeout
   767                small               small text
   772                source              origin, reference
   777                span                generic inline, span of text
   782                strong              important text
   787                summary             visible heading for details element
   792                target              goal, result, focus
   797                td                  table data cell
   802                th                  table header cell
   807                u                   underline
   812
   813    Document constants (general entity declarations)
   816                &bull;            • bullet
   820                &circle;          ∘ ring operator
   824                &emdash;          — emdash symbol
   828                &h1;                "h1" string
   832                &H1;                "H1" string
   836                &hsa;               "hsa" string
   840                &Hsa;               "Hsa" string
   844                &nbsp;              non-breaking space
  1034
  1035    END  h1.dtd

# 2023.05.30 22:12 # D:\kc\xt\h1\ # h1.dtd #
```

```
                                                               dawkIns .....
============================================================================ -->
```

Markup UK

Leveraging the Power of OpenAI and Schematron for Content Verification and Correction

Octavian Nadolu, Oxygen XML Editor

The purpose of this presentation is to provide an overview of what AI is, the potential benefits of using AI with Schematron and SQF for content verification and correction, and some of the challenges we face when using AI for this purpose.

AI (Artificial Intelligence) is a branch of computer science that studies and develops theories, methods, and technologies to allow machines to perceive, understand, and act in the world. AI is used to create algorithms that can learn, understand, and make decisions in complex environments. AI is used in many areas, including natural language processing, robotics, computer vision, data mining, and machine learning.

OpenAI is a research laboratory dedicated to developing and deploying AI in order to solve complex problems. OpenAI has developed algorithms that use reinforcement learning and deep learning to solve problems in robotics, natural language processing, and computer vision.

Schematron can be used to identify elements of a document and make assumptions about them, providing a powerful boost when used in conjunction with OpenAI algorithms. Using OpenAI algorithms with Schematron and SQF (Schematron Quick Fix) can help to automate the verification of the correctness, completeness, and accuracy of content. OpenAI algorithms can be used to automatically check and correct errors in content and markup. This can save time and money for content creators and publishers, as well as improving the accuracy of the content.

However, there are challenges associated with using OpenAI for content verification and correction. OpenAI algorithms may not accurately identify errors in content, and may not be able to make corrections that are appropriate for the context. Additionally, OpenAI algorithms may not be able to detect errors that are due to cultural or regional differences in language.

In conclusion, OpenAI can be a powerful tool for verifying and correcting content, but there are challenges associated with using OpenAI for this purpose. It is important to consider these challenges when using OpenAI for content verification and correction.

1. Introduction

In today's digital age, content verification and correction have become increasingly important, especially for businesses and organizations that rely on the accuracy and consistency of their content to communicate with their audiences. To address this need, many companies are turning to artificial intelligence (AI) and Schematron to automate the process of ensuring that their content meets certain standards and guidelines. Schematron, a powerful language for specifying rules for XML documents, allows developers to define their own custom messages and provide more descriptive feedback to users. By leveraging the power of AI and Schematron, businesses can streamline their content creation and editing processes and ensure that their messaging is clear, consistent, and error-free.

2. Artificial Intelligence

Artificial Intelligence (AI) refers to the ability of machines to perform tasks that typically require human intelligence, such as visual perception, speech recognition, decision-making, and language translation. AI is a broad field that encompasses a range of techniques, including machine learning, deep learning, natural language processing, and computer vision.

Machine learning is a subset of AI that involves training algorithms to learn patterns in data, without being explicitly programmed. Deep learning is a type of machine learning that uses neural networks with multiple layers to learn increasingly complex representations of data. Natural language processing is a subfield of AI that focuses on enabling machines to understand and generate human language. Computer vision is another subfield of AI that focuses on enabling machines to interpret and analyze visual information.

AI has numerous real-world applications across various domains, including healthcare, finance, transportation, customer service, and entertainment. As technology advances, AI continues to evolve and has the potential to revolutionize industries and enhance various aspects of human life.

3. Generative Pre-trained Transformer(GPT)

Generative Pre-trained Transformer (GPT) is a type of deep learning model that uses a transformer architecture to generate natural language text. The model is pre-trained on a large corpus of text data and then fine-tuned on specific tasks, such as language translation or text completion.

The transformer architecture is a type of neural network that is designed to process sequential data, such as text. It uses self-attention mechanisms to capture the relationships between different parts of a sequence, allowing it to generate more coherent and contextually appropriate text.

GPT is particularly useful for natural language processing tasks such as language translation, text summarization, and text completion. It has achieved state-of-the-art performance on a range of benchmarks, including the GLUE benchmark for natural language understanding and the COCO captioning challenge for image captioning.

Overall, GPT represents a significant advance in the field of natural language processing, enabling more accurate and effective text generation and understanding.

A transformer is a type of deep learning model architecture that is used for processing sequential data, such as natural language text. It was introduced in a 2017 paper by Vaswani et al. and has since become a popular choice for a wide range of natural language processing tasks.

The transformer architecture is based on the idea of self-attention, which allows the model to focus on different parts of the input sequence when making predictions. This is in contrast to traditional recurrent neural networks (RNNs), which process sequential data in a linear fashion and can struggle with long-term dependencies.

The transformer consists of an encoder and a decoder, each of which contains multiple layers of self-attention and feedforward neural networks. The encoder processes the input sequence, while the decoder generates the output sequence. During training, the model learns to predict the next token in the sequence based on the previous tokens, and can be fine-tuned for specific tasks such as language translation or text classification.

Overall, the transformer architecture has proven to be highly effective for natural language processing tasks, achieving state-of-the-art results on a range of benchmarks.

Embeddings, in the context of natural language processing (NLP) and machine learning, refer to the mathematical representations of words, sentences, or documents in a continuous vector space. Embeddings are used to capture the semantic meaning and relationships between words, allowing machines to understand and process human language.

Traditionally, words were represented as one-hot vectors, where each word in a vocabulary is assigned a unique binary vector with a dimension equal to the vocabulary size. However, one-hot vectors lack semantic information and are not suitable for machine learning algorithms that rely on numerical representations.

Embeddings address this limitation by mapping words to dense, lower-dimensional vectors in a continuous space. The goal is to encode similar words with similar embeddings, such that their spatial proximity reflects their semantic similarity. This is achieved through unsupervised learning algorithms, such as Word2Vec, GloVe, or fastText, which learn embeddings based on the context in which words appear in large corpora.

The resulting word embeddings can capture various linguistic relationships, such as word analogies (e.g., "king" - "man" + "woman" = "queen") and syntactic patterns. Additionally, word embeddings can be extended to represent larger units of text, such as sentences or documents, by aggregating the embeddings of constituent words.

Embeddings have become a fundamental component of many NLP tasks, including language translation, sentiment analysis, information retrieval, and text classification. They enable machine learning models to leverage the semantic information encoded in text and make more accurate predictions or understandings based on it.

4. Schematron and AI

Schematron is a schema language for XML documents that allows for more flexible and expressive validation than traditional XML schema languages like XML Schema and DTD. Schematron uses a rule-based approach to validation, allowing for complex business rules and constraints to be expressed in a clear and concise manner. It is often used in conjunction with other schema languages to provide more comprehensive validation of XML documents.

Schematron can be used with AI to automatically verify documents using an AI model. This can be done by integrating the AI model into the Schematron rules to check for specific criteria. For example, the AI model can check for the use of active/passive voice in the document and the Schematron rules can be set up to flag any instances where the wrong voice is used. Similarly, the AI model can check for adherence to a specific style guide and the Schematron rules can be set up to flag any deviations from that guide. The AI model can also check if the document answers a specific question and the Schematron rules can be set up to flag any instances where the question is not answered. Additionally, the AI model can check for spelling and grammar errors and the Schematron rules can be set up to flag any errors found by the model. Overall, integrating AI with Schematron can help to automate the verification process and ensure that documents meet specific criteria.

5. Schematron Quick Fix and AI

Schematron QuickFix (SQF) is a simple language that allows the Schematron developer to define actions that will correct the problems reported by Schematron rules. SQF was created as an extension of the Schematron language. It was developed within the W3C "Quick-Fix Support for XML Community Group". The first draft of the Schematron Quick Fix specification was published in April 2015, second draft in March 2018, and it is available on the W3C Quick-Fix Support for XML community group [https://www.w3.org/community/quickfix/] page.

Schematron Quick Fix (SQF) can be used with AI to automatically correct problems in documents. Here are some steps to use SQF with AI:

1. Create a set of rules using Schematron to identify problems in the document. These rules can include grammar and spelling checks, as well as more complex checks like sentence structure and coherence.
2. Use AI algorithms to generate suggestions for how to fix the identified problems. For example, an AI algorithm could suggest rephrasing a sentence to use active voice or to have a certain word count.
3. Implement the suggested fixes using SQF. SQF allows for user-defined fixes to be applied automatically, making it easy to correct problems in the document without manual intervention.
4. Test the document to ensure that the fixes have been applied correctly and that the document is now error-free.

By combining Schematron, AI, and SQF, it is possible to automate the process of correcting problems in documents, saving time and improving the overall quality of the document.

6. Implementation of AI in Schematron

To call an implementation of AI in Schematron, you can use XSLT functions or extension functions.

1. Using XSLT functions:

   You can use XSLT functions to call an implementation of AI in Schematron. For example, you can use implement function to call ChatGPT API. Here's an example:

   Example 1. Example of XSLT functions that calls ChatGPT

```
<xsl:function name="ai:chatGPT">
    <xsl:param name="userInput"/>
    <xsl:variable name="url" select="'https://api.chatgpt.com/v1/chatbot/question'"/>
    <xsl:variable name="requestBody" select="concat('{', '&quot;text&quot;:&quot;',
        $userInput,'&quot;', '}')"/>
    <xsl:variable name="response" select="document(concat($url, '?apiKey=',
        '&lt;your_api_key>'))//response"/>
    <xsl:sequence select="$response"/>
</xsl:function>
```

2. Using extension functions:

   You can also use extension functions to call an implementation of AI in Schematron. For example, you can define an extension function called `ai:verify-content` that takes an instruction and content as parameters and returns a boolean. Here's an example:

Example 2. Example an extensions function ai:verify-content()

```
<sch:rule context="...">
  <sch:assert test="ai:verify-content('instruction', 'content')">Message</sch:assert>
</sch:rule>
```

In this example, the `ai:verify-content` function is called within a `sch:assert` element to verify that the content meets certain criteria specified by the instruction.

To modify the content using AI, you can define an extension function called `ai:transform-content` that takes an instruction and content as string parameters and returns transformed content as string. You can use this function an SQF action to correct the content. Here's an example:

Example 3. Example an extension function ai:transform-content

```
<sqf:fix id="rephrase">
    <sqf:replace select="ai:transform-content('instruction', 'content')"/>
</sqf:fix>
```

The `ai:transform-content` function is called within a `sqf:replace` element and will transform the given content according to the instruction, the returned result will be inserted in the document replacing the current node content .

7. Examples of AI-driven Schematron and SQF Solutions

7.1. Check text consistency

One use case for Schematron using AI is to check if the text in an XML document is consistent and easy to read and understand. For example, you may want to ensure that the text is written in plain language, uses simple sentence structures, and avoids technical jargon or complex terminology. To achieve this, you can define a Schematron rule that checks for specific patterns or structures in the text. For instance, you may define a rule that checks if the sentences are too long or if the text contains too many complex words or phrases. If the rule detects any issues, it can report an error or a warning to the user, indicating the specific location of the problem.

Example 4. Example a Schematron rule that verifies if the text is easy to read and understand

```
<sch:rule context="p">
    <sch:assert test="ai:verify-content('Is the text easy to read and understand?', .)">
        The text in not easy to read and understand</sch:assert>
</sch:rule>
```

Once you have identified the issues in the text, you can use SQF (Schematron Quick Fixes) to provide suggestions or corrections to the user. For example, you may provide a suggestion to correct the text to be easy to read and understand by calling the ai:transform-content() function.

Example 5. SQF fix that corrects the text to be easy to read and understand

```
<sqf:fix id="rephrase">
    <sqf:description>
        <sqf:title>Correct the consistency of the text</sqf:title>
    </sqf:description>
    <sqf:replace match="text()" select="ai:transform-content(
        'Correct the text to be easy to read and understand', .)"/>
</sqf:fix>
```

7.2. Check text voice

In this example, we want to create a rule that verifies if the text voice is active. This means that the text should be written in a way that emphasizes the subject of the sentence and uses active verbs. For example, instead of saying "The ball was thrown by John", we should say "John threw the ball".

Example 6. Rule that verifies if the text voice is active

```
<sch:rule context="shortdesc">
    <sch:assert test="ai:verify-content('Is active voice used?', .)">
        In the description we should use active voice.</sch:assert>
</sch:rule>
```

If the text does not follow this rule, we can create a fix that reformulates the text to use active voice. This will improve the clarity and readability of the text.

Example 7. SQF fix that that reformulates the text to use active voice

```
<sqf:fix id="rephrase">
    <sqf:description>
        <sqf:title>Reformulate the text to use active voice</sqf:title>
    </sqf:description>
    <sqf:replace match="text()" select="ai:transform-content('
        Reformulate to use active voice', .)"/>
</sqf:fix>
```

7.3. Answer to question

Another example is a rule that checks if the text answers to a specified question. Let's say we have a text that discusses OpenAI, but we want to make sure that the text actually answers the question "What is OpenAI?"

Example 8. Rule that verifies if the text does answer to a specific question

```
<sch:rule context="p[@id='openai']">
    <sch:assert test="ai:verify-content('Does it answers to the question: What is OpenAI?', .)">
        The test does not answer to the question "What is OpenAPI?" </sch:assert>
</sch:rule>
```

If the text passes this rule, we can be reasonably confident that it does answer the question "What is OpenAI?"

However, if the text does not pass the rule, we know that it may not be providing a clear answer to the question. In this case, we might need to revise or restructure the text to make sure it does answer the question.

Example 9. SQF fix that that reformulates the text to answer to the question

```
<sqf:fix id="rephrase">
    <sqf:description>
```

```
        <sqf:title>Reformulate the text to answer to the question:
                        What is OpenAI?</sqf:title>
    </sqf:description>
    <sqf:replace match="text()" select="ai:transform-content(
        'Reformulate the text to answer to the question: What is OpenAI?', .)"/>
</sqf:fix>
```

7.4. Check the number of words

In this case, we want to create a rule that checks the number of words in the description element of an XML document. The rule should ensure that the description contains less than 50 words. If the description contains more than 50 words, the rule should fail and report an error. The implementation uses XPath functions to tokenize the string and count the words.

Example 10. Rule that verifies the number of words from the shortdesc element

```
<sch:rule context="shortdesc">
    <sch:report test="count(tokenize(.,'\s+')) > 50">
         The description must contain less than 50 words.</sch:report>
</sch:rule>
```

You can also provide a fix for the error by suggesting a corrected version of the description that contains less than 50 words. The AI can help you to do this by reformulating the phrase.

Example 11. SQF fix that reformulates the phrase to have less than 50 words

```
<sqf:fix id="rephrase">
    <sqf:description>
        <sqf:title>Reformulate phrase to contain less than 50 words</sqf:title>
    </sqf:description>
    <sqf:replace match="text()" select="ai:transform-content(
           'Reformulate phrase to contain less than 50 words', .)"/>
</sqf:fix>
```

7.5. Check if block of text should be a list

This Schematron rule is designed to check if a block of text should be converted to a list. It can be used to ensure that content is properly formatted and structured for readability and accessibility.

Example 12. Rule that verifies the text from a paragraph should be converted to a list

```
<sch:rule context="p">
    <sch:report test="contains(., '- ')">
        The text should be converted to a list</sch:report>
</sch:rule>
```

If the rule detects that the text should be converted to a list, it can trigger a fix that generates an unordered or ordered list from the set of phrases. This can help to improve the organization and clarity of the content, making it easier for readers to understand and follow.

Example 13. SQF fix that creates a list from a set of phrases

```
<sqf:fix id="replace">
    <sqf:description>
        <sqf:title>Create a list from the phrases from the paragraph</sqf:title>
    </sqf:description>
    <sqf:replace match="text()">
        <xsl:value-of select="ai:transform-content(
            'Create a Dita unorderd list with an item from each phrase', .)"
            disable-output-escaping="yes"/>
    </sqf:replace>
</sqf:fix>
```

If the generated content contains markup you need to use the *disable-output-escaping="yes"* in order to insert the markup unescaped in the document.

7.6. User-Entry - Check technical terms

Technical terms are specialized words or phrases that are used within a particular field or industry. When writing technical documents, it's important to ensure that these terms are explained adequately so that readers who may not be familiar with them can understand the content. One way to check if technical terms are explained adequately is to use a Schematron rule that interrogates an AI . The rule might look something like this:

Example 14. Rule that verifies if the technical terms are explained adequately

```
<sch:report test="ai:verify-content('Are the technical terms explained ambiguous?', .)"
       The text uses technical terms that are not explained adequately.</sch:report>
```

To fix this issue, the user could provide an explanation for the technical term. However, sometimes it may be necessary to reformulate the phrase to make it more understandable. In this case, a fix could be added to the Schematron rule that allows the user to specify how to reformulate the phrase.

Example 15. SQF fix that allows the user to specify the prompt that will be send to the AI

```
<sqf:fix id="reformulateUser">
    <sqf:description>
        <sqf:title>Specify how to reformulate the phrase</sqf:title>
    </sqf:description>
    <sqf:user-entry name="userInput" default="'
         Reformulate phrase and replace the ambiguous terms with a more accurate one'">
        <sqf:description>sqf:title>How to correct:</sqf:title>sqf:description>
    </sqf:user-entry>
    <sqf:replace match="text()" select="ai:transform-content($userInput, .)"/>
</sqf:fix>
```

The user can specify the instruction for the fix in the sqf:user-entry. A default value instruction is proposed to the user, but the user can decide do correct the phrase providing other instruction to the AI system (such as: explain term, define term, replace term with a new one)

8. Generate Fix Automatically

When a Schematron rule fails during validation, it generates an error message indicating the context of the error. However, it does not provide a fix for the issue. The fix for the error message can be generated using AI, which analyzes the context of the rule and suggests a correction to the XML document. This can help to automate the process of fixing errors in XML documents and save time for developers.

Example 16. Example: A Schematron rule that verifies the number of words from shortdesc element

```
<sch:rule context="shortdesc">
    <sch:report test="count(tokenize(.,'\s+')) > 50">
         The description must contain less than 50 words.</sch:report>
</sch:rule>
```

You can use the Schematron rule to generate a fix using AI. The content to correct is provided in the context of the Schematron rule, and the goal is to use AI to generate a new text that will serve as a fix for the issue. Specifically, the AI should generate a new text to replace the content of the "shortdesc" element, which should contain a brief description of the issue.

This approach can be extended beyond Schematron to other types of error messages in software development. By using AI to generate fixes automatically, developers can save time and focus on more complex tasks, while also improving the overall quality of their code.

Of course, there are some challenges to this approach. For example, the message needs to be as concise as possible so that the AI can use it effectively. Additionally, sometimes the context of the message may need to be modified or additional operations may need to be performed to generate an effective fix. However, overall this approach can save a lot of time and effort compared to manually creating

9. Develop Schematron using AI

Artificial intelligence can be used to develop Schematron rules. Machine learning algorithms can be trained to recognize these patterns and generate Schematron rules that can be used to validate future XML documents.

The user can provide a prompt that describes the Schematron rule. He can describe a specific scenario or use case for which the Schematron rules will be created using AI. The AI will generate the rules based on the scenario he provides.

For example the user can provide a prompt like: "A Schematron assert that verifies the number of words to be 10". The generated content by the AI can be something like this:

```
<sch:assert test="count(tokenize(., '\s+')) = 10">There should be exactly 10.</sch:assert>
```

Another example of prompt can be to verify if the text contains an email. I this case the prompt can be something like: "A Schematron assert that verifies if there is an email in text". The content generated by the AI can be something like this:

```
<sch:assert test="matches(., '\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b')">
                There is no email in the text</sch:assert>
```

10. Conclusion

Schematron is an XML-based language used for validating the content of XML documents. It allows users to define rules that can be used to verify the correctness and completeness of the content. With the help of AI, Schematron can be made even more powerful.

One way to use AI with Schematron is to use machine learning algorithms to analyze the content of XML documents and identify patterns and anomalies. This can help to detect errors and inconsistencies that might otherwise be missed by traditional validation techniques. For example, AI can be used to identify semantic errors, such as incorrect usage of terms or concepts, that might not be caught by simple syntactic validation.

Another way to use AI with Schematron is to use SQF fixes to correct errors automatically. For example, if an XML document contains a misspelled word, an AI-powered Schematron can automatically correct the spelling without requiring any manual intervention. This can save time and effort and improve the accuracy of the validation process.

In addition, AI can be used to generate fixes automatically. For example, if an XML document contains an error that can be corrected automatically, an AI-powered Schematron can generate the corrected version of the document automatically. This can save time and effort and improve the overall quality of the content.

As AI technology continues to improve and evolve, Schematron can be further developed to take advantage of these advancements. For example, new machine learning algorithms can be integrated into Schematron to improve its ability to detect errors and anomalies.

Overall, the use of AI with Schematron has the potential to revolutionize the way content is validated and corrected. By leveraging the power of AI, Schematron can provide more accurate and efficient validation, leading to higher quality content and improved user experiences. As AI technology continues to grow and improve, the possibilities for Schematron are endless.

Markup UK

XQS: A Native XQuery Schematron Implementation

Andrew Sales

This paper will cover XQuery for Schematron (XQS) (pron. /ɛksˈkjuːz/),[1] a Schematron processor being implemented in native – and naïve – XQuery. To the author's knowledge, there are no complete implementations publicly available at this time.[2] The purpose of the work is at least two-fold: to demonstrate the utility of an XQuery query language binding for arguably the primary quality assurance technology applicable to XML, and to provide a second "reference" implementation while work on the latest revision of the ISO standard proceeds. Although not required by the standardization process, it clearly helps to be able to answer the question standards authors should consider when new features present themselves: How would you implement that?

1. Background

The existence of a Schematron validator implemented in XQuery has been the subject of periodic enquiry in the community.[3] Respondents on mailing lists typically indicate the existing and highly serviceable XQuery wrappers around an XSLT implementation, such as are available for the main XML databases.[schematron-existdb][schematron-basex][schematron-ML]

While Schematron has been implemented in a range of other languages, apparently the only members of the XML technology stack to fulfil this role so far have been XPath, such as ph-schematron[ph-schematron] (in its "pure" variant) and the author's defunct XMLProbe, later open-sourced as Probatron,[probatron][4] and XSLT.[skeleton][schxslt] Indeed, to many Schematron users, the XSLT implementations *are* Schematron to all intents and purposes. It is a mark of the success of those implementations that they have come to be regarded in this way.

There may however be advantages to pursuing a native XQuery implementation. The standard itself still defines a range of query language bindings (QLBs)[iso-qlbs], some of which are moribund or of questionable longer-term value[5]. There are QLBs for XPath and XSLT in all their current versions defined normatively, and while the values `xquery`, `xquery3` and `xquery31` are all reserved for future use, these are not defined at all.

For the purposes of updating the standard,[6] the present work has obvious benefits in informing what a QLB for XQuery should look like. For the end user, it also gives them options not previously available, most obviously the native integration of a Schematron

[1] https://github.com/AndrewSales/XQS
[2] Though note the experimental work in this area by David Maus: https://github.com/dmj/schematron-xquery.
[3] See e.g. http://x-query.com/pipermail/talk/2011-November/003704.html
[4] The later updates archived at https://code.google.com/archive/p/probatron4j/ and https://code.google.com/archive/p/probatrondotnet/ are wrappers around the skeleton implementation.
[5] E.g. STX and EXSLT, as raised in https://github.com/Schematron/schematron-enhancement-proposals/issues/26.

schema into an XQuery-based validation workflow, e.g. in a native XML database, or, as is the case with ph-schematron[ph-schematron], the possiblity to isolate and address individual schema components. Further potential benefits of this to the XQuery user are discussed in more detail below.

This paper will examine three main areas arising from the work:

- ◇ design goals and decisions made
- ◇ implementation details of note
- ◇ how these may influence the next edition of ISO Schematron, with regard to the XQuery QLB.

Some lessons learned in these attempts will be expanded on below, and some of the remaining work highlighted.

2. Rationale

Unlike some other bodies, ISO [https://www.iso.org/home.html] does not require any "reference" implementation to be developed in support of a standard it publishes. The so-called skeleton implementation was mothballed as no longer maintained in late 2020, and the main XSLT implementation at present is SchXslt [https://github.com/schxslt/schxslt]. Once ISO approved the work for a new edition of the standard, it seemed useful to have another implementation, in a query language not as yet defined by the standard, to support and inform this work. The author has also been a keen advocate of further Schematron implementations at previous community meetups[meetups], and a further implementation of course introduces more choice and options for the schema author and end user alike.

3. Design goals

The high-level brief was to produce an implementation conforming to ISO/IEC 19757-3:2020. Conformance is defined in Clause 7 of the standard. What it terms *simple conformance* amounts to reporting whether an XML document is valid to a given Schematron schema: a simple pass/fail, with SVRL[7] output not required. *Full conformance* is also defined, but does not expand much on that, adding as criteria correct attribute values and a proscription on duplicate variable names; SVRL is not mandated here either.[8] To meet perhaps most users' expectations, align with SchXslt, as well as provide the means to compare outputs with that tool's, SVRL was deemed a necessity from the start; text output is out of scope.

An additional reason for choosing XQuery was the appeal of relatively rapid development facilitated by its concise and declarative syntax.

3.1. Conformance over performance

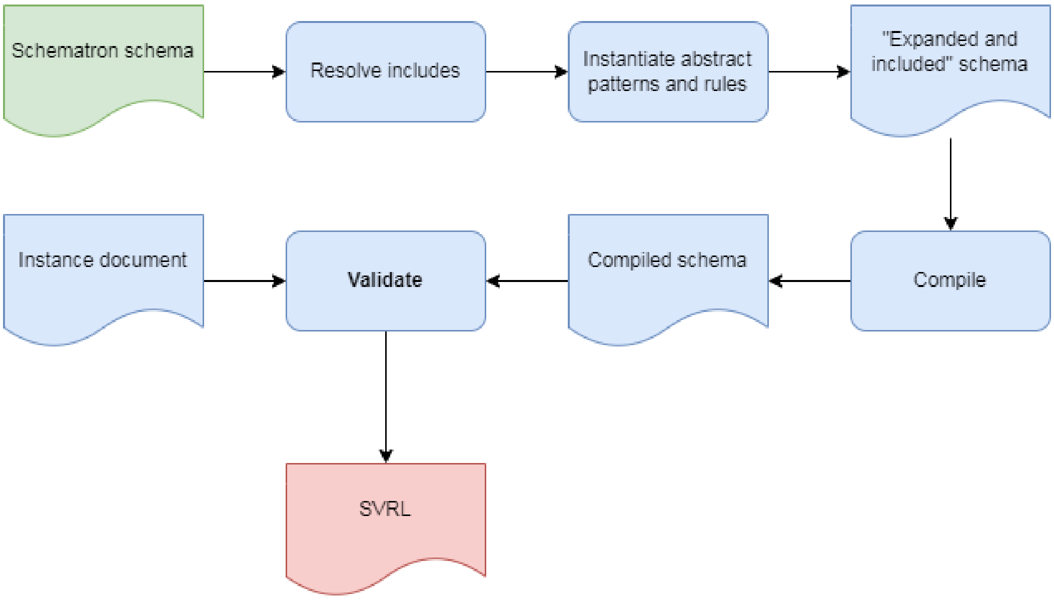
Since the work is being carried out to support development of the standard, the primary goal is to be conformant. It is not intended to be production-grade software in terms of performance. The implementation is naïve in this sense: no (conscious) optimization has been done in order to improve performance at this stage.

[6] ISO/IEC JTC 1/SC 34 approved a new revision of the standard and convened a new working group to carry this out in September 2022.
[7] Schematron Validation Reporting Language, defined in Annex D (informative).
[8] Technically, it could not be, since its definition in the standard is not normative.

3.2. Dynamic evaluation

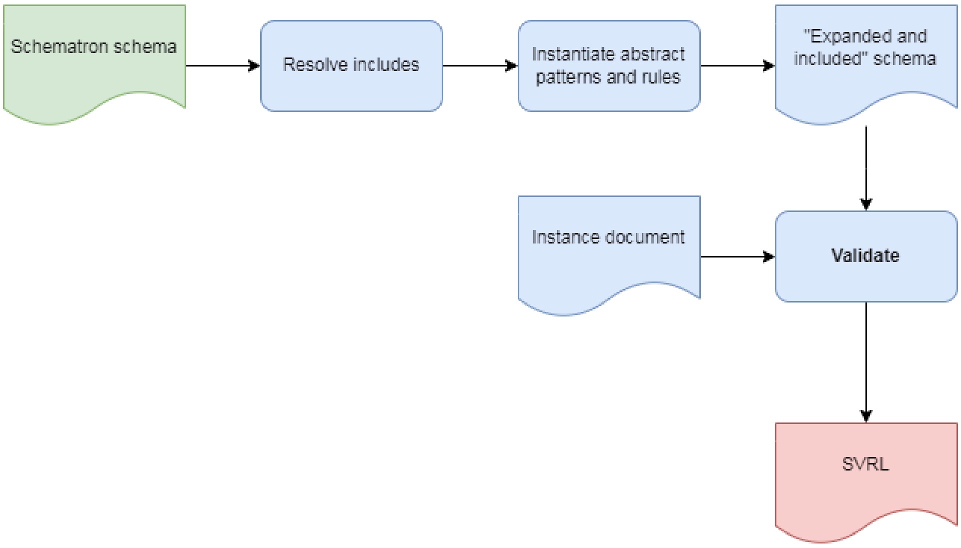
In the XSLT implementations, processing a Schematron schema includes a "compilation" step, whereby the schema is turned into a transform.

Figure 1. Schematron processing steps: expansion, inclusion and compilation



The prospect of invoking validation directly – without any so-called schema "compilation" step – was also attractive, based on the experience of previous work on XMLProbe and Probatron, which used an XPath engine[jaxen] to evaluate expressions without any intermediate steps. When a user invokes Schematron validation in an IDE such as oXygen[9], they are admittedly unaware of the three-step process of resolving includes, instantiating abstract rules and patterns, and producing a compiled schema in the form of an XSLT transform — but doing it this way would be an interesting exercise nonetheless.

Figure 2. Schematron processing: "dynamic evaluation"


Schematron schema
Resolve includes
Instantiate abstract patterns and rules
"Expanded and included" schema
Instance document
Validate
SVRL

[9] https://www.oxygenxml.com/

3.3. Portability

A further objective was the possibility of writing this tool in such a way that validation could be run directly, or using the schema compiled as standard XQuery, in an engine-agnostic way, i.e. that the code should be portable as far as possible.

4. Caveats

This section sets out some points to bear in mind about the implementation at the present time.

4.1. Expansion and inclusion

The dynamic evaluation portion of the code was written first, to establish proof of concept. The next step was intended to be resolution of includes and expansion of abstract constructs, before turning to schema compilation. However, at that point it seemed that the greater challenge, and arguably more interesting feature of the work, lay in compiling to XQuery, so expansion and inclusion have been deferred for now[10]. In the meantime, users should rely on other tools (such as SchXslt) to perform these pre-processing steps if needed.

4.2. "Native"?

For reasons of familiarity in the main, BaseX[11] was chosen as the development platform, so the code uses its dialect of XQuery 3.1. This means that dynamic validation relies on that, but a generated XQuery ("compiled schema") uses no engine-specific functions or syntax, so also runs under e.g. Saxon[12]. See also Section 10.2 [156].

4.3. Mandated XQuery QLB

Note that an XQuery query language binding is required by XQS and the application terminates with an error if this is not present.

5. Approach

This section documents some key design decisions.

5.1. Context is everything

5.1.1. Document level

An early consideration was what input the processor should be expected to handle when validating. The text of the standard speaks about "an XML document" (Clause 1, Scope) and refers to "the instance document" in several places. The fact that the text is written in this way does not explicitly preclude batch processing or multiple context items, but Schematron was clearly conceived of initially as a validator consuming a single XML document as input. By contrast, using XQuery means we have its in-built facility to address a corpus of multiple documents, such as may be stored in a database or on a file system.

XQS adheres to the strict interpretation of the standard for now: the input is expected to be a single document node. This is in line with the other QLBs, but there may be a case for specifying the XQuery binding to allow multiple input documents, e.g. as returned by a call to `collection()`.

[10] See https://github.com/AndrewSales/XQS/issues/6.

[11] https://basex.org/

[12] See https://www.saxonica.com/.

5.1.2. Node level

A Schematron schema uses `rule/@context` to select nodes in a document to apply assertions (represented by `assert` and `report`) to. So in contrast to the rule-based processing of XSLT (where `rule/@context` typically translates to `xsl:template/@match`), with XQuery we are selecting nodes we are interested in, rather than them being supplied to us by default. So the decision here was to interpret any expression given in `rule/@context` as being evaluated in the context of the instance document root.

The obvious but important implication for schema authors who are used to targeting XSLT implementations is that e.g. `<rule context="*">` will *only match the root element* in XQS, as opposed to any element in the document (and therefore the path in this case should become `//*` instead).

5.1.3. Assertion level

Once a `rule/@context` matches (or the rule "fires", as the standard has it), the assertions contained in the `rule` are applied to each item returned.

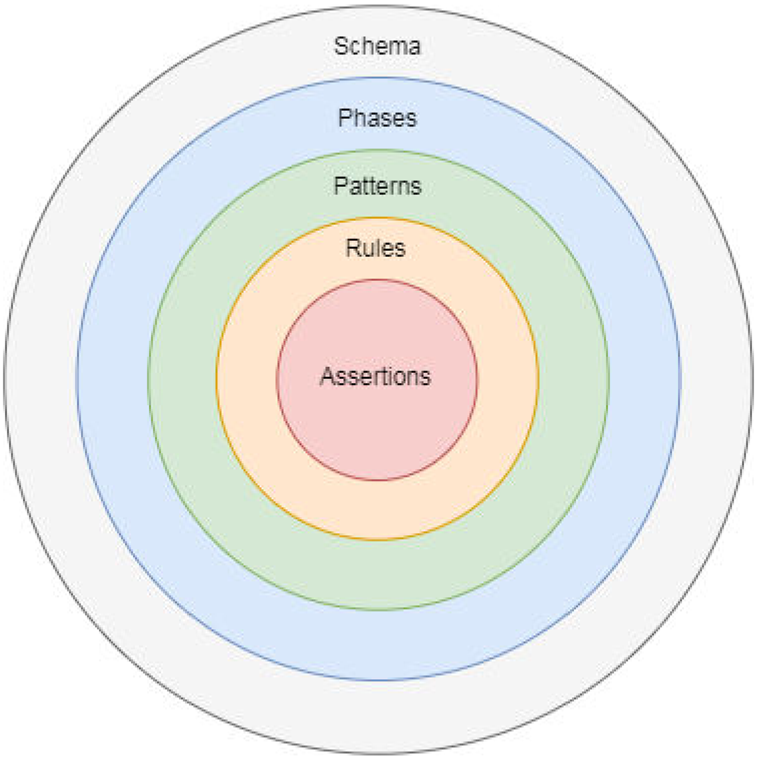
The existing XPath and XSLT QLBs specify that the rule context may be one or more nodes, so the XQuery binding in this implementation makes the same assumption.

6. Implementation

The codebase for XQS encapsulates two approaches to processing a Schematron schema for validation purposes. Dynamic evaluation, as mentioned above, processes the instructions the schema contains in memory. The other method of compiling a schema produces an XQuery main module, which can be serialized and used for validation, either at the command line or as part of a larger application.

The code for each approach is organized into the respective modules `evaluate.xqm` [https://github.com/AndrewSales/XQS/blob/main/evaluate.xqm] and `compile.xqm` [https://github.com/AndrewSales/XQS/blob/main/compile.xqm]. In broad terms, they each take the same approach to the detail of processing the schema, with successive functions to handle in turn the schema, phases, patterns, rules and assertions.

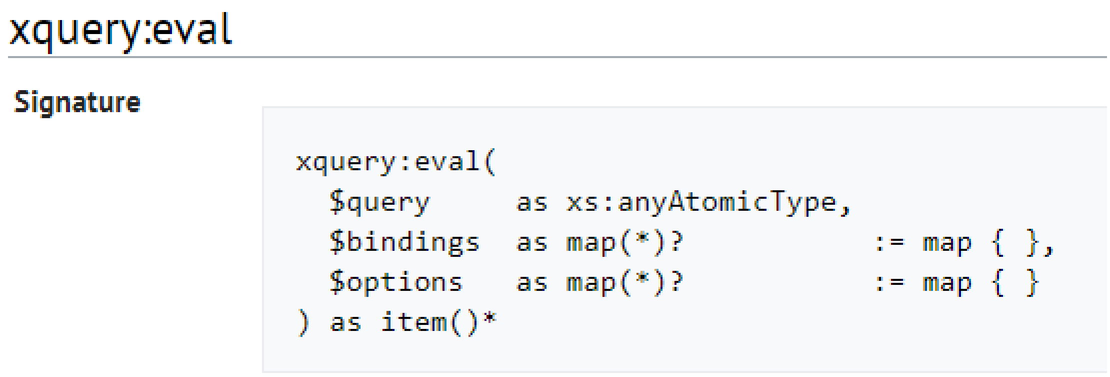
Figure 3. Schematron schema structure



As might be expected, the main difference between evaluating and compiling is that the evaluation functions are part of the codebase, while those in the compiled schema are generated automatically by functions which are themselves part of the codebase. The following sections outline some implementation details of note.

6.1. Dynamically evaluated schema

This portion of the tool makes use of the BaseX-specific `xquery:eval()` function to evaluate XQuery expressions dynamically:



xquery:eval

Signature

```
xquery:eval(
  $query      as xs:anyAtomicType,
  $bindings   as map(*)?            := map { },
  $options    as map(*)?            := map { }
) as item()*
```

When evaluating the rule context, a string of the query is first constructed, containing the prolog, consisting of namespace declarations and local variables, followed by the expression to be evaluated. The second argument to `xquery:eval()` here is a map of global variable bindings, which are effectively passed in as external variables, and the context, i.e. the instance document. BaseX binds this by the convention of using the empty string as a key.

The validation context comprising namespaces, variable bindings and user-defined functions is maintained via a map constructed when the schema is first parsed. This must be updated at two points during evaluation: while evaluating `patterns`, and assertions. In the case of assertions, this simply means changing the evaluation context to any rule context returned. The handling of `patterns` is slightly more involved.

6.1.1. Evaluating `patterns`

Patterns are essentially a grouping mechanism for `rules`. They can contain variables (represented by `let`), which are unusual in that they are local in scope to the containing `pattern`, but are evaluated in the context of the instance document root.[13] This change in scope but not evaluation context means the map of bindings must be updated for every `pattern`.

This behaviour differs from that of the XSLT implementations.[14]

6.1.2. The `documents` attribute

Another feature of `patterns` is the `documents` attribute, introduced in the 2016 edition of the standard. It affords the convenience of applying `rules` to the "subordinate" documents located at the URIs derived by evaluating `documents` in the context of the instance document root, for example:

```
<sch:pattern documents="/element/@secondary">
  <sch:rule context="/">
    <sch:report test="root"/>
  </sch:rule>
</sch:pattern>
```

Here, each document located at the URIs in `/element/@secondary` would have the `rule` applied to it, and any root element `root` reported.

There are two issues with this shifting of context to other documents, which the standard arguably underspecifies. One is that it is only implicit that the `rules` in that `pattern` should only apply to the subordinate documents retrieved (which is what XQS assumes) and not also to the instance document itself. The other (an edge case, but possible) is whether `pattern` variables should be in scope when evaluating the `documents` attribute [15]; XQS assumes that they are.

6.1.3. Rule processing

The if-then-else behaviour of `rules`, whereby no further `rules` are processed once one has fired, is implemented using simple tail recursion:

```
declare function eval:rules(
  $rules as element(sch:rule)*,
  $prolog as xs:string?,
  $context as map(*)
)
as element()*
{
  if(empty($rules))
```

[13] This differs from `rule` variables, which are evaluated in the context of the nodes returned by evaluating `rule/@context`.

[14] See https://github.com/schxslt/schxslt/issues/135 for details regarding the behaviour of SchXslt here.

[15] As in `<pattern documents='/subordinate/documents[not(@path = $these)]'>`, where `$these` is defined as a variable within that `pattern`.

```
  then ()
  else
    let $result := eval:rule(head($rules), $prolog, $context)
    return if($result)
    then $result
    else eval:rules(tail($rules), $prolog, $context)
};
```

where the result of the `eval:rule()` call will be the empty sequence if no matching rule context exists.

6.1.4. Advisory notes

One aspect of this processing approach is that syntax errors can lurk undetected in a branch of code that is not reached, by analogy with interpreted languages such as Python. For example rules or assertions can have incorrect syntax in their `context` or `test` attribute, for example:

```
<sch:rule context="/..">
    <sch:report test="???">[...]</sch:report>
</sch:rule>
```

One remedy would be for XQS to do some static analysis ahead of time, by parsing these expressions to see if they are valid XQuery; BaseX provides `xquery:parse()`, which could be used for this purpose.

One potential drawback of the dynamic evaluation approach worth mentioning if deployed to validate documents in a BaseX database is that `xquery:eval()` imposes a global read lock.[16] If this is an issue for a given workflow, it would be advisable to use a compiled schema instead.

6.2. Compiled schema

In some respects, this generated code approach is more straightforward to develop, not least because its tangible output is code which can itself then be debugged.

Each `pattern`, `rule` and assertion generates a function (in the `local` namespace), with evaluation context and local variable scope more simply managed at the function level.

The processing of `rules` is again handled by tail recursion, but this time the function consumes a sequence of function items, for variation:

```
declare function local:rules($rules as function(*)*)
as element()*
{
if(empty($rules))
  then ()
  else
    let $result := head($rules)()
    return if($result)
    then $result
    else local:rules(tail($rules))
};
```

There is no need to pass the instance document to each function representing a rule, since it is passed in as an external variable to the module. Where subordinate

[16] See https://docs.basex.org/wiki/Transaction_Management.

documents are present due to `pattern/@documents` (see Section 6.1.2 [152]), an alternate version of this function is also generated with the signature `declare function local:rules($rules as function(*)*, $doc as document-node()*)`.

When it comes to serializing the compiled schema, since most of the generated functions can return some SVRL markup, for simplicity a sequence of strings and elements is passed to `serialize()`, with the custom `basex` serialization method specified.

> ### Note
>
> The compiled schema places variables for internal use in the XQS namespace, to avoid clashes with variable names in the source schema. For this reason, it is inadvisable to declare the XQS namespace (`http://www.andrewsales.com/ns/xqs`) in a schema to be compiled by XQS.

7. Other features

7.1. User-defined functions

By analogy with the use of `xsl:function` under the XSLT QLBs, user-defined XQuery functions are supported by XQS. These will be included for use in the (evaluated or compiled) schema when placed before the patterns, using element `function` in the XQuery namespace, `http://www.w3.org/2012/xquery`, e.g.

```
<sch:schema>
  <sch:ns prefix='myfunc' uri='xyz'/>
  <function xmlns='http://www.w3.org/2012/xquery'>
  declare function myfunc:test($arg as xs:string) as xs:string{$arg};
  </function>
  <sch:pattern>
    <sch:rule context="/">
      <sch:report test="root"><sch:value-of select='myfunc:test(name(root))'/></sch:report>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

7.2. Evaluating schema components

The API of the evaluation module contains discrete functions for schema, patterns, rules and assertions. This means that a complete schema is not required in order to perform evaluation of these components in isolation. One possible and likely application of this technique would be unit testing. The following is an example taken from the XQS test suite itself:

```
(:~ active pattern processed :)
declare %unit:test function _:process-pattern()
{
  let $result := eval:pattern(
    <sch:pattern id='e' name='f' role='g'>
      <sch:rule context='*' id='a' name='b' role='c' flag='d'/>
    </sch:pattern>,
    map{'instance':document{<foo/>}, 'globals':map{}}
  )
  return unit:assert-equals(
    $result,
    (<svrl:active-pattern id='e' name='f' role='g'/>,
    <svrl:fired-rule context='*' id='a' name='b' role='c' flag='d'/>)
  )
};
```

Here, the call to `eval:pattern()` (shown in bold) evaluates the `pattern` passed in against the context provided by the second, map argument. In this instance, the unit test is to check that patterns and rules are reported correctly in SVRL, but the user can extend this approach elsewhere via calls to `eval:rule()` and `eval:assertion()`; further examples of this appear in the XQS test suite.[17]

7.3. Maps, arrays and anonymous functions as variables

At the time of writing, work is in progress to implement assigning these types to variable values. While their datatype may be discoverable by the processor (again, via `xquery:parse()`, which conveniently returns an XML representation of the parsed tokens and their types), including support for these lends weight to the argument for adding optional attribute `as` to Schematron's `let` to express the datatype, which should also remove the need to support `xsl:variable` in an XSLT implementation.

8. Unit testing

On the subject of unit testing, the BaseX built-in `unit` module was used extensively during development. The tests for evaluation were written first, in support of that module, but this meant that the compile tests did not need to be written from scratch and could be adopted with little adjustment.

The Schematron conformance suite[18] is a set of XML documents containing instances and schemas and their expected behaviour under a conformant implementation. An XQuery tool was written to generate unit tests from these assets.

9. Evaluation

The design goals at the outset of this work could be considered to have largely been met:

◇ the application is pure XQuery

◇ on conformance, it passes the tests in the core conformance suite that it can be expected to pass

◇ dynamic evaluation has been implemented

◇ testing so far has shown that compiled schemas run (and produce the same output) under Saxon as well as BaseX.

10. Status of the work

It should be noted that this is still an early release, and should be treated as such: it is not considered production-ready, owing to lack of testing with real-world schemas and the noted focus on conformance over performance.

10.1. The conformance suite

Some of the conformance suite tests relate to expansion and inclusion, others are XSLT-specific, so these have been excluded from testing.

[17] https://github.com/AndrewSales/XQS/tree/main/test
[18] https://github.com/Schematron/schematron-conformance

In addition, the implementation of `pattern` variables does not match that of SchXslt, as mentioned above, so that test has been modified in XQS.

10.2. Future work

A list of enhancements is kept here [https://github.com/AndrewSales/XQS/issues?q=is%3Aopen+is%3Aissue+label%3Aenhancement]. Expansion and inclusion are a priority, so that the application can be used with schemas displaying those features, without recourse to other tools. It is also planned to develop dynamic evaluation to run under other engines. An ultimate objective is to have the complete codebase engine-agnostic, as far as possible.

XSLT implementations boast an API to provide callbacks when producing SVRL. No consideration has been given to this in XQS so far, but it is worth exploring in the interests of parity.

Bibliography

[schematron-existdb] https://github.com/Schematron/schematron-exist

[schematron-basex] https://github.com/Schematron/schematron-basex

[schematron-ML] https://github.com/ndw/ML-Schematron

[probatron] Probatron [https://code.google.com/archive/p/probatron/]

[ph-schematron] ph-schematron [https://phax.github.io/ph-schematron/]

[skeleton] Skeleton implementation [https://github.com/Schematron/schematron], archived October 2020.

[schxslt] SchXslt [https://github.com/schxslt/schxslt]

[iso-qlbs] ISO/IEC 19757-3:2020, Clause 6.4; Annexes H-M. https://www.iso.org/standard/74515.html.

[meetups] Schematron Users Meetup, at XML Prague 2017-20, 2022, https://www.xmlprague.cz/

[jaxen] Jaxen [https://github.com/jaxen-xpath/jaxen]

Markup UK

Quality in Formatted Documents

Tony Graham, Antenna House

“Markup quality assurance” is the theme for Markup UK 2023, but what does “quality” mean when the markup is meant to be formatted for human consumption? When that markup is transformed from an original document in a completely different XML vocabulary? This presentation looks at different ways of assessing or assuring the quality of both the markup and the formatted document.

1. Markup Quality

1.1. XSL-FO

focheck, available on GitHub and bundled with Oxygen, provides the best available way to check, create, and edit XSL-FO markup.

A distinctive feature of XSL-FO is that most of the properties’ values can be expressions that the XSL formatter evaluates when it formats the document. This, of course, stymies grammar-based checking of an XSL-FO document because the datatype of a property value can’t be known until the expression is evaluated.

focheck combines a RELAX NG schema for checking the structure of an XSL-FO file with Schematron and a REx-generated parser for evaluating property value expressions. The Schematron also checks relationships between elements that can’t be expressed in a grammar. When used as an Oxygen framework, the included Schematron Quick-Fixes allow interactive fixing of some of the errors common in XSL-FO documents.

1.2. CSS

HTML can, of course, be checked using any number of HTML-aware editors and linters. However, formatting systems typically provide extension properties and functions that aren’t supported by browsers. Some of these ‘extensions’ are properties originally defined in CSS Working Drafts but dropped from later versions.

Extensions are, by definition, not universally supported, and the same is true for their editing and validation. The CSS editor in Oxygen, for example, recognises the extensions for Oxygen Chemistry, but not the extensions for other formatters. An Oxygen framework cannot extend the syntax checking of the Oxygen CSS editor, so Antenna House provides a VSCode extension that recognises the Antenna House CSS Formatter extension properties (but not the Oxygen extensions).

The VSCode extension supports syntax highlighting of the Antenna House extensions and provides tooltips that show a property’s definition and allowed values plus links to the property’s documentation and code samples.

2. Formatted Quality

2.1. Regression testing

Sometimes the most important quality check that you can make is to ensure that a change that you made does not have any unintended consequences. Regression testing is as important in document formatting as it is in any other software field.

When the markup is deeply nested and contains multiple interacting properties, or the properties are defined in separate stylesheets containing rules that cascade in ways that may not be immediately obvious, then checking the formatted output for changes can be better use of your time than looking at diffs of files and trying to imagine the effects.

Antenna House developed a visual regression testing system for its own use that was then released as a product. It can compare single raster images, multi-page documents, or directories of documents and produce both summary reports and reports showing the exact visual differences between two pages. The system rasterises its source documents at a user-defined resolution and compares them pixel-by-pixel. As such, the sources do not have to be from Antenna House Formatter and, in fact, could be generated by two disparate sources.

2.2. Automated analysis

Languages, including English, have stylistic conventions for formatted text. The origins of the conventions may be for readability, for aesthetics, for commercial reasons, or for a mix of these. Some are now just considered to be good design without reference to the underlying reason. Books on typography or book design will usually cover a subset of possible problems, but even the reference books differ in what they consider to be a problem, the threshold for a condition becoming a problem, and even the terminology for describing a problem.

Automated analysis, introduced in Antenna House Formatter V7.0 and expanded in V7.1, can detect a range of error conditions:

- Too many blank pages at the end of the document

  The printing and binding method used for a book may require that the book is a multiple of 8, 16, 32, or even more pages. Extensions to the force-page-count property make this possible with AH Formatter V7.1. However, the forced page count can result in empty pages at the end of the document just to fulfil the requirement. Empty pages are a cost to the publisher with little or no obvious benefit.

- Too many consecutive lines end with a hyphen

  Too many consecutive lines that end with a hyphen increase the likelihood that a reader will either skip reading a line or read the same line twice. Both the Chicago Manual of Style (17th edition) and Elements of Typographic Style recommend a maximum of three consecutive lines that end with a hyphen.

- Too many consecutive lines that all start or all end with the same word

  This is similar to the problem with multiple consecutive lines that end on a hyphen. Multiple consecutive lines that start with the same word or multiple lines that end with the same word can result in a reader either skipping a line of text or rereading a line. The Chicago Manual of Style (17th edition) recommends a maximum of three lines that either start or end with the same word. Book Typography warns against multiple lines that end with the same word but does not provide a limit and does not mention lines that start with the same word.

◇ Lines before or after current block

When a chapter does not start on a new page, there can be a requirement for a minimum number of lines either before or after the chapter heading. Book Typography recommends at least three lines above and below the chapter heading. This can usually be enforced using the widows and orphans properties, but not when, for example, the previous chapter ends with short lines of dialogue.

◇ Page widow

A short last line of a block of text that is formatted as the first line on a page or column can affect readability.

◇ Paragraph widow

A short last line of a block of text can affect readability. A secondary consideration is that many paragraph widows can add extra pages, and cost, to a document.

◇ River

A river occurs where spaces on consecutive lines overlap, or nearly overlap. Rivers are more likely to occur in justified text than in text that is aligned to one side or is centred. A large or long river of white-space may interfere with comprehension of the text. People differ in their sensitivity to rivers, but it is often noted as problem for people with certain cognitive disabilities, including dyslexia.

◇ Unbalanced spread

It can be an aesthetic requirement that text blocks on facing pages are the same length.

◇ White-space

Excessive white-space between words can affect readability.

The problems found by the automated analysis are reported as log messages. The Antenna House ‘analysis-utility’ project on GitHub provides scripts to process the error log and the document to generate either an analysis report or a copy of the formatted document that is annotated to show the locations of the errors.

2.3. PDF/UA checking and remediation

Tools for checking conformance to the requirements of PDF/UA range from free tools that check a file at a time to utilities that monitor your network and check every PDF file. Similarly, tools for remediating (repairing) PDF/UA files range from free tools to systems that are “price on application”.

A Dependency Management Approach for Document and Data Transformation Projects

Jorge Sánchez

This talk introduces and evaluates the suitability of Apache Ivy as a dependency manager for document and data transformation projects, which is less common in these projects. The proposed solution will provide an optimized set of templates based on the development of Arousa and test common scenarios like dealing with local repositories and setting up a shared network repository.

The approach has shown that it is suitable for dependency management on XSLT and XProc transformation projects, and the code samples will be shared for future reference. The initial experience with Arousa seems to back the validity of the solution, and the templates provided can help with the added complexity.

1. Introduction

I decided to prepare Arousa after a few conversations here and in Prague. I remember talking about using Ivy for Xml dependency management but the idea never seemed to catch up.

Arousa is both a tool and a set of templates for managing dependencies in document transformation projects using Apache Ant and Apache Ivy. As a tool, Arousa provides a script for managing template projects with a prebuilt integration with Ivy. The provided templates try to serve as a set of examples to ease start with Ivy in this type of projects.

The purpose of this document is to evaluate Ivy as a suitable dependency manager for document and data transformation projects.

I'll use Arousa to introduce Ivy and as a mean to simplify the adoption. We'll iterate over Ivy concepts, their possible use cases and check the plausibility. We'll refer to the prepared examples and at the end we'll evaluate the results and conclusions.

2. Introducing Apache Ivy

Apache Ivy is the de facto dependency manager for Apache Ant. Apache Ivy is meant to be flexible (can even be used without Ant) and compatible with Maven and other type of file and network based repositories. Ivy is managed with Xml files, just like Ant, content management technologies like Xslt, Xproc and lots of data sets and content documents.

Ivy is not restricted to jar or package files, it can handle arbitrary types of artifacts, trace elaborated configuration chains and using multiple repositories.

3. Starting with Arousa

Arousa is wrapped as a bundle, with a managing script and a set of ant template projects integrated with Ivy. You can add your templates taking the existing ones as an example.

Arousa helps you during the creation and configuration of the Ant projects and eases certain calls but all the Arousa magic is made by standard Ant and Ivy capabilities. Once the projects are created, they are self contained Ant projects. You can invoke them with Arousa or directly using Ant commands.

The Arousa configuration is done, in a similar way to a JVM or Ant, by setting its HOME installation path and adding the arousa script to the executable path.

```
export AROUSA_HOME=<Your_Installaton_path>
export PATH=$PATH:$AROUSA_HOME
```

Arousa main script require a bash environment (you may use Cygwin on Windows) Apache Ant and Java. Depending on the specific project some internal variables may also need adjustment, like the Xproc engine path for example.

After it is configured, you can run the Arousa script and print the list of available options.

```
$arousa
 Arousa Dependency Script
 Syntax : arousa command argument1 argument2 ..

 Arguments :

 - template-project <name> : Builds a new project arguments
              <template-project-name(optional)> <project-name>

 - template-project-content : gets the template project content

 - template-project-list : prints a list of the templates

 - update-dependencies : Updates dependencies

 - publish-dependency : Publish the main dependency

 - update-main-dependency : Get the repository version
                                        of the main dependency

 - clean-cache : Cleans ivy project dependency cache

 - template-project-for-dependency : Get a template project
                                  for the indicated dependency

- update-arousa-configuration : Update arousa configuration files
```

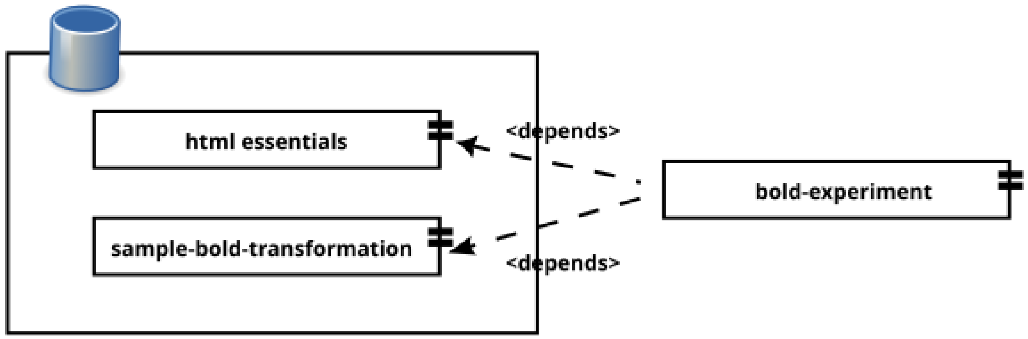
The following command, for example, prints the available project templates.

```
$ arousa template-project-list
```

4. A bold experiment. How much time/effort does it take?

We are going to introduce Ivy and Arousa with a basic example. The purpose is to evaluate how easy it is to work with Arousa and how much effort involve manage dependencies with Ivy.

The example will use a pair of external xslts, one that convert the text nodes to bold, and other that wraps the content into html.



We are going to open the bold experiment folder from the Arousa MARKUPUK23 examples and remove the bold-experiment subfolder.

You can navigate now to the sample-bold-transformation folder and run:

```
$ arousa publish-dependency
```

At this point you probably have published the first ivy dependency. We are going to create now a project, using the default template, and name it “bold-experiment” with the following command.

```
$ arousa template-project bold-experiment
```

In this example we need to adjust the path of the repository (in order to use the previously published example dependencies). Modify the “demo.samples.repository.root” value of the conf/arousa-ivy-settings.xml to fit your location of the examples path, it should look similar to this:

```
<property
        name="demo.samples.repository.root"
        value="C:/Users/john/arousaws/arousa-examples/
                                    bold experiment/repository" />
```

Now we are going to add the dependencies to the conf/arousa-ivy.xml file.

```
<dependency org="org.markupuk.2x023.arousa.samples"
              name="sample-bold-transformation" rev="1.0" >
      <artifact name="sample-bold-transformation"
                  type="packaged" ext="zip" />
    </dependency>

    <dependency org="net.vionta.templates.html"
                            name="html-essentials" rev="1.0" >
      <artifact name="html-essentials" type="packaged"
                ext="zip" />
</dependency>
```

You could instead just copy the file (from the commands notes folder).

```
cp commands/arousa-ivy.xml bold-experiment/conf/arousa-ivy.xml
```

After adding the dependencies declaration, and setting the repository path, you can retrieve the dependencies with the following command.

```
$ arousa update-dependencies
```

You should see on the screen the Ivy report while it is retrieving the dependencies. At this point, the dependencies should show on the deps folder of your project. We are ready to add the Xslt steps to the build target of the build.xml.

```
<xslt style="deps/sample-bold-transformation/xsl/bold-demo.xsl"
      basedir="test/data"
      destdir="build"
      includes="*.xml"
      extension=".xml"
    />

<xslt style="deps/html-essentials/xsl/html-report-basic-structure.xsl"
      basedir="build"
      destdir="build"
      includes="*.xml"
      extension=".html"
    />
```

Now we can add some test data to a file.

```
$ mkdir -p test/data
$ echo '<div>This text will be bold</div>' > test/data/basic_test.xml
```

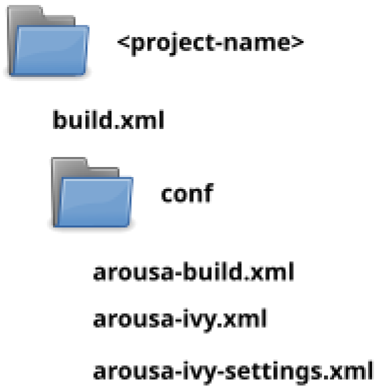
And finally run the build target.

```
$ ant build
```

And that’s all, we have prepared a small project, configured Ivy, declared and retrieved the dependencies and executed the transformations.

5. Introducing the Arousa project structure

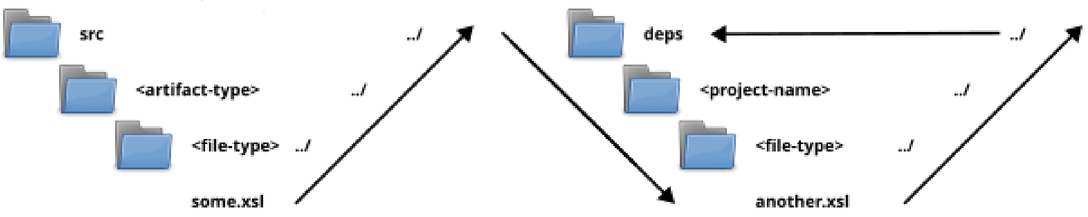
Arousa creates a project structure with a base build.xml file, source folders, etc. You may have noticed the conf folder. The conf folder contains the usual ivy files prefixed with the “arousa-” key. It creates the arousa-ivy that describe the dependencies, the arousa-ivy-settings that describes the repositories and an arousa-build.xml with the ant tasks used in the dependency management operations. The base build file calls imports and calls the arousa build tasks.



In order to use the same path references to dependencies from the source folders and from the dependency folders we use a folder naming convention. The idea is to have the same depth of folders both source folders and from the dependencies folders.

```
./src/<dependency-type>/xsl   -->     deps/<some_project>/xsl
./src/<dependency-type>/xproc -->     deps/<some_project>/xproc
```

Doing so you can call some script, like for example a xsl:import on some.xsl with the same path ../../../deps/<project-name>/<file-type>/some.xsl both from the source folder and later on when the xsl is deployed to other projects deps folder.



Note that you could avoid this convention, using options, string substitutions, etc.

6. Ivy abstraction. Introducing artifact types

Ivy is a quite abstract tool, in the sense that is focused on its tasks without getting into irrelevant details. This is probably one of its biggest adoption difficulties, as it lacks specific guidelines on some of its elements.

For example, what is a dependency type? What should it be used for? How? Is it related to the file extension?

Ivy does not care much about what do you use an artifact type for. As long as something is a type for you, Ivy handles it accordingly. It can be aligned with the file extension or not. We must remark here that Ivy has a specific attribute for the artifact file extensions. That extension attribute take the artifact type value if its not specified explicitly.Ivy does not care much about what do you use an artifact type for. As long as something is a type for you, Ivy handles it accordingly. It can be aligned with the file extension or not. We must remark here that Ivy has a specific attribute for the artifact file extensions. That extension attribute take the artifact type value if its not specified explicitly.

The Arousa template projects package files using zip archives. We use to refer to them as “transformations” because we use it for transformation sources. You are free to refer to it with the name that make more sense for you. The template projects adds the content from the doc folder to the published packages to make some sort of documentation available in the dependent projects. It may be useful for sharing details about some css rules, or maybe how should we call xslts, their names and use cases, etc.

There may be cases where the documentation contents may need to be handled with different delivery cycles. We are going to use this use case to illustrate artifact types.

We have added an example (dita-sources example) where we share several dita source sets chained. If you check for example the dita-sources-1 project you may notice a modification of the arousa build where we pack a doc zip explicitly.

```
<target name="arousa-package-doc"  if="${doc.folder.present}" >
 <mkdir dir="temp-doc" />
 <mkdir dir="temp-doc/doc" />
 <mkdir dir="temp-doc/doc/${arousa.project.name}" />
 <copy todir="temp-doc/doc/${arousa.project.name}" >
 <fileset dir="doc" >
 <include name="**" />
 <exclude name="deps/**" />
 </fileset>
 </copy>
 <!-- .......... Creating temporary doc ........... -->
 <zip destfile="dist/${arousa.project.name}-doc.zip" >
 <zipfileset dir="temp-doc"  >
 <include name="**" />
 </zipfileset>
 </zip>
 <delete dir="temp-doc" failonerror="false" />
 </target>
```

We have also added a complementary publication to the conf/arousa-ivy.xml

```
<publications>
    <artifact name="dita-sources-1" type="transformations"
              ext="zip" conf="default"/>
    <artifact name="dita-sources-1-doc" type="documentation"
              ext="zip" conf="default"/>
</publications>
```

We’ve also added the type to the repository pattern in the conf/arousa-ivy-settings.xml.

```
<property name="demo-arousa-pattern"
        value="[organisation]/[module]/[type]/[revision]/
                                  [module]-[revision].[ext]"
        override="true" />
```

Once the configurations have been adjusted, and some contents have been added to the doc folder, you can pack the contents and publish the examples. We are going to use the ant targets directly with the following command.

```
$ ant package dist
```

At this point you can retrieve and use the contents from a second or third project. Take a look at the conf/arousa-ivy.xml second dita-sources project. We have added the dependency with only the documentation artifact.

```
<dependencies defaultconf="default">
    <dependency org="org.markupuk.examples.vionta"
                        name="dita-sources-1" rev="1.0" >
    <artifact name="dita-sources-1-doc" type="documentation"
                                              ext="zip" />
</dependency>
 ...
```

If you type the following command you should bring the dependency to your project.

```
$ arousa update-dependencies
```

This example shows an interesting point about what can and should we use dependency management for. We can use dependency management for software binaries, code, etc. but we can also describe elements, like content, datasets and shared resources. There are several cases where this approach can be used, like for example the legal clauses that may be added to each contract, proposal, books, etc. in a publishing or in a rfp process. Usually interrelated elements that our process depend on and are shared as resources on a timely basis, both managed by different groups or used in different contexts.

7. A Key difference with Maven

A key difference between maven and Ivy is that maven nature is intended for a single module output. Ivy is designed instead for multiple output artifacts from the same component.

Maven is probably more suitable for the intermediate, most developer centric processes (structured activities) while and Ant is useful in the last steps of delivery, where you may need to add specific tasks, consider configuration and adjustments depending on user needs, client requirements, device configuration, different channels, etc.

Obviously there are newer alternatives, but Ant and Ivy, due to the shared Xml format with Xslt, Xproc and content formats like dita, docbook, etc. seems like the natural match for the job.

8. Configuration chains

If you had doubts with dependency types, you may have the same doubts with configurations. What should I use Ivy configurations for?

In a similar way as artifact types, anything that makes sense for you as configuration, and obviously fit into something that can be managed with artifacts by Ivy, should be a valid configuration for Ivy.

Ivy configurations are a powerful concept. Each component can have multiple configurations that can be arranged using inheritance. But Ivy configurations don’t stop here, as each configuration may depend on other configurations that can be adjusted for each artifact. The Ivy capabilities to describe dependency configuration rules is outstanding.

We are going to illustrate how can we take advantage of Ivy configurations with a simple example. We are going to take into consideration a report. A report that may have to be sent on a timely basis from a ticketing system like Jira or Redmine.

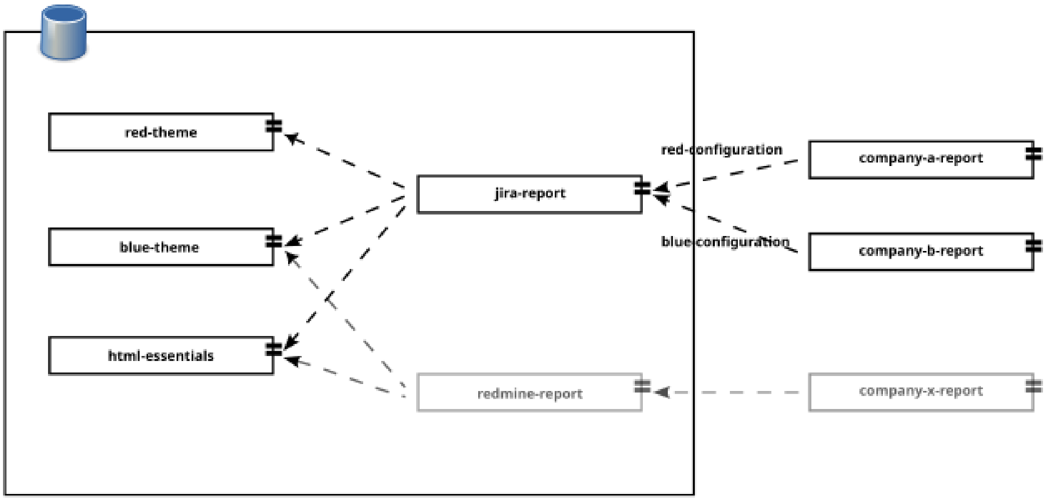
On a functional level it may not be perceived as convenient as both solutions have their own reporting capabilities, but at the end there may be roles that need visibility on the team performance that usually don't have access or don't have time to access some platform on a private VPN or DaaS.

The example will have the following key components, although there are some other smaller ones not listed here.

- A graphical theme, with css files and icons.

- A Jira set of transformation stylesheets and utilities.

- The current data projects, where we are going to perform the Jira or Redmine issue list analysis.



At this point, a question may arise, should our report depend on a specific theme project for each report style/colour set or should we use a specific project with several output configurations? We are going to start with two different theme projects, a red one and a blue one.

Both options are fine (one project with two configurations or two project managed with configurations), and you may choose the strategy that best fits or describes your problem. We will try to illustrate both strategies with the Jira report on our examples

The configuration-jira-report folder from our examples has two themes the blue-theme-report and the red-report-theme. Both projects contain some graphic resources and css that are used in the usual web folders.

The jira report project is going to depend on the theme projects but it's not going to use them directly. The Jira report project has two different configurations, not surprisingly named red and blue report. We are going to use this project only to develop, test and evolve the Jira Xslts and Xprocs.

```
<configurations>
    <conf name="default" />
    <conf name="red-report" extends="default" />
    <conf name="blue-report" extends="default" />
  </configurations>

  <publications>
```

```
    <artifact name="jira-report" type="transformations"
              ext="zip" conf="*"/>
  </publications>

  <dependencies defaultconf="default">
    <dependency org="org.markupuk.examples.arousa"
                name="blue-report-theme" rev="1.0"
                      conf="blue-report->default" >
      <artifact name="blue-report-theme"
                type="transformations" ext="zip" />
    </dependency>
    <dependency org="org.markupuk.examples.arousa"
                name="red-report-theme" rev="1.0"
                conf="red-report->default" >
      <artifact name="red-report-theme"
                type="transformations" ext="zip" />
    </dependency>
    <dependency org="org.markupuk.examples.arousa"
                name="html-essentials" rev="1.0" >
      <artifact name="html-essentials"
                type="transformations" ext="zip" />
    </dependency>
  </dependencies>
```

Once we need to build the actual reports, retrieve the issue list from the Jira API, select a part of the data, etc. we are going create a specific project. This type of projects are going to depend on one of the possible configurations (red or blue).

We are going to call our example project company-a-report. The following lines show its ivy dependency description file.

```
<ivy-module version="1.1">
  <info organisation="org.markupuk.examples.arousa"
    module="companya-report"/>
  <configurations>
    <conf name="default" />
  </configurations>
  <publications>
    <artifact name="companya-report" type="transformations"
              ext="zip" conf="default"/>
  </publications>
  <dependencies defaultconf="default">
    <dependency org="net.vionta.reports.jira"
                name="jira-report" rev="1.0"
                conf="default->red-report" >
      <artifact name="jira-report" type="transformations"
                ext="zip" />
    </dependency>
</dependencies>
</ivy-module>
```

You may notice the conf attribute (conf="default->red-report"). This attribute indicates that this project default configuration depends on the jira report “red-report” configuration. If you check on the “jira-report” project dependencies, the red report configuration depends on the red report theme. Also the red report configuration extends the default configuration which brings shared dependencies between both versions.

The example is a very simple demonstration of the Ivy configuration capabilities. Ivy can be used to describe and manage really complex configuration interrelations.

9. The Ivy Cycle.

Ivy brings the dependencies using a combination of two steps, resolve and retrieve.

- The resolve step checks the dependency graph, tries to locate the files on the repositories and brings the artifacts to your cache.

- The second one is the actual retrieval, from the local Ivy cache to the project location.

It is important to keep this in mind when you publish new versions of the artifacts. Depending on the configurations you may need to clean the cache to force the resolve step again.

If you try to retrieve, you expect to get newer artifact versions, and you keep getting the same ones you use the following command.

```
$ arousa clean-cache
```

10. Ivy flexibility (resolvers)

Ivy relies on resolvers and resolver chains to locate and retrieve artifacts. The possibility to define several types of repositories in chains, with different setups, makes artifact management really flexible.

The approach also eases the possibility to cooperate between groups in a decentralized manner. Exposing repositories to other groups with shared folders, web servers, ftp servers, etc.

Repositories and resolvers are configured in the ivy settings file, that within Arousa it is named conf/arousa-ivy-settings.xml.

The list of Ivy resolver types is quite extensive, including a ibiblio/maven type.

The following are ibiblio repository declarations that can be used to handle java jars.

```
<ibiblio name="ibiblio" m2compatible="true" />
```

```
<ibiblio name="jboss2" m2compatible="true"
root="http://repository.jboss.com/maven2/" />
```

In many of our examples, like the bold experiments we have prepared a specific repository only for the test using a folder location.

```
<property name="demo.samples.repository.root"
          value="C:/Users/jorges/ws/arousaws/arousa-examples/
                  repositories" override="false"/>
 <property name="demo-arousa-pattern"
        value="[organisation]/[module]/[revision]/
                  [module]-[revision].[ext]"
        override="true" />
   <resolvers>
    <chain name="demo">
      <filesystem name="arousa-demo" m2compatible="true" >
    <ivy pattern="${demo.samples.repository.root}/
                                  ${demo-arousa-pattern}"/>
    <artifact pattern="${demo.samples.repository.root}/
                                  ${demo-arousa-pattern}"/>
      </filesystem>
    </chain>
...
```

Another usual repository type is the http/url one. In the dual resolver example conf/arousa-ivy-settings.xml you can find the following url example.

```
<resolvers>
  <dual name="dual-example">
    <filesystem name="ivys">
      <ivy pattern="${dual-example-repository-path}/
                        [module]-ivy-[revision].xml"/>
     </filesystem>
     <url name="two-patterns-example">
    <artifact pattern="http://aiweb.cs.washington.edu/
                    research/projects/xmltk/xmldata/data/
                                [module]/[artifact].[ext]"/>
     </url>
   </dual>
</resolvers>
```

11. Working with “Others”, the Dual resolvers

A dual resolver is one where we can combine two different repositories, one for the artifacts and other for the descriptors (arousa-ivy.xml files).

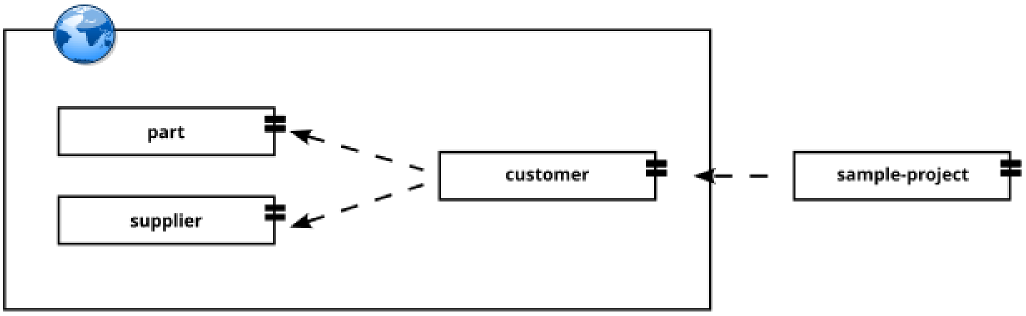
We can use it, for example, when we don’t actually manage the artifact repository (we use an external one) but we need to use It with our own set of references that we can store somewhere else.

Dual resolvers open a very interesting possibility, since other teams may not be willing to adopt practices like dependency management but it is easy to adhere to certain shared naming convention between teams when sharing resources.

With Dual resolvers, as long as the artifacts are placed using a naming convention in a shared network resource you can arrange the ivy metadata by yourself.

We are going to demonstrate this capability using the dual-resolver-data example. In this example we are going to relate three dataset files from an external resource.

We are going to take the customer, part and supplier dataset files from:



http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/

In our ivy files we describe the dependency from the customer ivy file to the part and supply ivy file:

```
<ivy-module version="1.1">
  <info organisation="edu.washington.cs.aiweb.tpc-h"
    revision="1.0" module="tpc-h"/>
  <configurations>
    <conf name="default" />
  </configurations>
  <publications>
    <artifact name="customer" type="data" ext="xml"
              conf="default" />
  </publications>
  <dependencies defaultconf="default">
    <dependency org="edu.washington.cs.aiweb.tpc-h"
                name="tpc-h" rev="1.0" >
      <artifact name="part" type="data" ext="xml" />
    </dependency>
    <dependency org="edu.washington.cs.aiweb.tpc-h"
                name="tpc-h" rev="1.0" >
      <artifact name="supplier" type="data" ext="xml" />
    </dependency>
  </dependencies>
</ivy-module>
```

You can verify that we have added to our internal repository the three ivy files. Also we’ve configured the dual resolver pointing to the remote washinton.edu site.

```
<?xml version="1.0" encoding="UTF-8"?>
<ivysettings>
  <property
    name="dual-example-repository-path"
value="${user.home}/dev/ws/arousaws/arousa-examples/examples/
                                  dual-resolver-data/repository"
       override="true" />
    <settings defaultResolver="dual-example"/>
    <resolvers>
    <dual name="dual-example">
      <filesystem name="ivys">
    <ivy pattern="${dual-example-repository-path}/
                                  [module]-ivy-[revision].xml"/>
      </filesystem>
      <url name="two-patterns-example">
    <artifact pattern="http://aiweb.cs.washington.edu/
     research/projects/xmltk/xmldata/data/[module]/[artifact].[ext]"/>
      </url>
    </dual>
  </resolvers>
</ivysettings>
```

The first part of the dual resolver points at the ivy descriptors location while the second one points to the artifacts. We call the update dependencies with the following command.

```
$ arousa update-dependencies
```

The call does not only resolve the customer dataset described in the project dependencies, but, as in our repository the three files are described as related, Ivy bring us the three necessary files.

```
<ivy-module version="1.1">
  <info organisation="org.markupuk.arousa.examples.data-download"
    module="data-download-test"/>

  <configurations>
    <conf name="default" />
  </configurations>

  <publications>
  </publications>
   <dependencies defaultconf="default">
    <dependency org="edu.washington.cs.aiweb.tpc-h"
                name="tpc-h" rev="1.0" >
      <artifact name="customer" type="data" ext="xml" />
    </dependency>
  </dependencies>
</ivy-module>
```

12. A step further

It is not common to expose datasets using dependency management. There are obvious reasons to avoid this approach with transactional or frequently updated interrelated data.

On the other side, databases tend to become performance bottlenecks on many information systems. The re-utilization of shared optimized datasets seems appropriate when:

- The resources are updated on a timely basis using datasets, with different data cycles.

- The datasets are shared. They are generated or published by different teams or may be used in different contexts.

- We are not interested on a specific the timing. The data can be retrieved periodically, daily or weekly for example.

- The volume or the performance of the datasets is not challenging.

Take for example a retail company app that handles sales, payments, inventory and delivery. We need to treat the transactional data, like capturing sales and payments information with extreme care. On the other side there are recurring reports like and information that may be shared, reducing CPU cycles and network calls, as a complementary option to ETLs, DataHubs, etc.

Another interesting option could be to enrich browser side scripts with preloaded options data, which could reduce the number of network round trips and accelerate load times.

13. An advanced example

You may probably be willing to have a simple dependency declaration directly on the Xslts and Xprocs. We didn't choose that approach because that means that we would have to declare every small dependency between each file. That would also exclude from the artifacts graphic or web resources, specially the binary ones.

If you have a very specific context where you work mainly with Xslts and Xprocs we could adjust the solution to manage dependencies directly.

In the advanced-configuration example we have added two projects, one that publishes an xslt and other that depends on it. We have added a namespace alias (“arimp”) to identify the dependency declarations in both xslts. The “build-tasks-report.xsl” has an element that indicates that the xslt should be published.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:arimp="http://www.vionta.net/schemas/arousa-import/
                                   v1.0/arousa-import.xsd"
    exclude-result-prefixes="arimp"
>
 <arimp:publication artifact-name="build-tasks-report"   />
  <xsl:template match="/">
...
```

On the target project the actual report file has a similar declaration indicating the dependency.

```
<xsl:stylesheet
    version="3.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:arimp="http://www.vionta.net/schemas/arousa-import/
                                   v1.0/arousa-import.xsd"
    exclude-result-prefixes="arimp"
    >
  <arimp:dependency
      component="lineal-fods" org="net.vionta.
                               transformation.spreadsheet"
      rev="1.0" artifact="build-tasks-report" />
  <xsl:template match="/">
...
```

Both projects are Xproc capable and include an xproc on the conf folder called generate-ivy-dependency-file.xpl.

```
<p:declare-step
    xmlns:p="http://www.w3.org/ns/xproc"
    version="3.0">

 <p:input port="source" primary="true" > <p:inline> <documento/>
                                          </p:inline> </p:input>
 <p:output port="result" primary="true" pipe="result@final-list" />
 <p:load name="load-ivy-file"  >
   <p:with-option name="href" select="'./arousa-ivy.xml'" />
 </p:load>
 <p:directory-list name="xslt-directory-list"
       include-filter=".xsl"  exclude-filter=".xsl~" >
   <p:with-option name="path" select="'../src/xsl/'" />
 </p:directory-list>

 <p:for-each name="xsl-files" >
   <p:with-input select="//*:file[@*:name]"  />
   <p:load name="load-file-content"  >
     <p:with-option name="href"
          select="concat('../src/xsl/', .//@*:name)" />
   </p:load>
 </p:for-each>
 <p:wrap-sequence name="xslt-imports" wrapper="xsl-files" >
 </p:wrap-sequence>
 <p:insert name="merge" position="first-child" >
   <p:with-input port="insertion"  pipe="result@load-ivy-file" />
 </p:insert>

  <p:xslt name="final-list">
      <p:with-input port="stylesheet"
            href="./extract-import-elements.xsl"/>
  </p:xslt>

   <p:store name="final-serialization" >
     <p:with-option name="href" select="'./arousa-ivy.xml'" />
   </p:store>
 </p:declare-step>
```

As you can see, the Xproc lists the xslts from the src/xsl folder and wraps them with the current arousa-ivy file. The xslt merges the organization and configuration options from the existing ivy file with the imports extracted from the xslts.

As a result, the script populates the Ivy dependency file with the dependency relations.

```
<?xml version="1.0" encoding="utf-8"?>
<ivy-module version="1.1">
 <info organisation="org.markupuk.examples.arousa"
       module="sample-fods-report"/>
  <dependencies defaultconf="default">
      <dependency org="net.vionta.transformation.spreadsheet"
                  name="lineal-fods" rev="1.0">
         <artifact name="build-tasks-report"
                   type="transformation" ext="xsl"/>
      </dependency>
   </dependencies>
</ivy-module>
```

Both the xsl publishing and retrieval works with the same commands as the rest of examples (publish-dependency and update-dependencies).

14. Conclusions

Ivy fits naturally with Ant, Xslt, Xprocs, and can handle easily graphic resource packages. Our experience reinforces the perception that it blends well in this context.

It certainly adds a some complexity to the build process, as we need to manage additional files and configurations. In some cases, where the number of files or components is small it may not be worth it. There may be cases, where shared resources managed with discipline may be more than enough.

The solution has a learning curve, and the number of training materials are limited.

Even thought, the point where the benefits overcome the initial difficulties can be reached easily. The dependency management helps to maintain the code well structured and organized. Also, since each functionality can be managed in an isolated project, with test data, the code can be easily organized and maintained.

As an advice it is better to consider the adoption design in advance on environments with certain complexity. It is better than make corrections once there are several dependencies and developed projects.

In general, the number of dependencies managed and the benefits from the practice exceeded our initial expectations. It helps to avoid non DRY practices and there’s the tendency to segregate the functionalities. In general, after some initial training it is fairly easy to add more components.

We have increased the initial folder depth (from two levels to three), adding the project name to the exploded folder, as the number of dependencies started to make difficult to trace which file came from which component.

Ivy flexibility is remarkable, we’ve looked for potential problems in during the preparation of the examples or flaws that made this approach not appropriate. All the proposed examples and capabilities worked fine and the integration were easier and faster than expected.

Bibliography

[SAMPLES] Demo Exapmles. The demo examples described in this talk are placed at: https://github.com/vionta/MarkupUK2023

[ANT] Apache Ant. https://ant.apache.org

[IVY] Apache Ivy. https://ant.apache.org/ivy

[Xslt] Xslt Standard. https://www.w3.org/Style/XSL

[Saxon] Saxonica Xslt. https://www.saxonica.com

[Xproc] Xproc Standard. https://www.w3.org/TR/xproc

[Morgana] Morgana Xproc. https://www.xml-project.com/morganaxproc-iiise.html

[Aiweb] Sample dataset catalogs (Aiweb.cs.washinton.edu) http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/

[EXPath-Pkg] EXPath Pkg main site. http://expath.org/modules/pkg/

[EXPath-Pkg-Spec] EXPath Pkg Specification. http://expath.org/spec/pkg

[maven-catalogBuilder-plugin] Christoph Marchand Catalog Builder. https://github.com/cmarchand/maven-catalogBuilder-plugin/wiki

Markup UK

Tool-Based Transformations

Use the Markup, Luke

Liam Quin, Delightful Computing

This paper discusses the development of tools to explore and maintain XSLT, XQuery, and other markup-processing code. The tools described include FreqX, Xarcissus, and Eddie 2, which help to explore sample documents and large DTDs. The paper also covers XSLT DTD-coverage testing and encourages others to take a similar approach in developing tools to explore markup.

1. Introduction

Maybe you have to work on some existing XSLT or XQuery code, whether you wrote it years (or minutes) ago or whether someone else wrote it. Maybe you have to write new transformations or queries.

You have some exploring and some planning to do.

You need to explore the possible input. If you are lucky, maybe there is *actual* input for you to explore and, later, use for testing. Maybe there is pre-existing code to explore; after you have started work, soon there will be new code to explore!

All of these things can be done with or without tools beyond a simple text editor, an XML well-formedness checker, an XSLT or XQuery processor, and so on. But if the input is marked up, we can explore the markup, and maybe tools, whether off-the-shelf or custom, can make that much easier.

A strength of XML and especially XSLT is enabling people who do not think of themselves as programmers to do advanced text processing. It follows that you do not need to be an experienced programmer to write useful tools.

2. Project steps and phases

Although the actual break-down varies between projects and with different methodologies, the following tasks are always involved with a project involving marked-up documents:

- Requirements Analysis: What do we think we need?
- Data Analysis: What do we have?
- Strategy: How will we get there?
- Status Review: Where are we now?
- Implementation: Let’s go there!

◇ Testing and Fixes: Are we there yet?

In a waterfall model these steps were performed in order, with no feedback from one to another. This enabled fixed-priced projects but unfortunately led to systems that did not meet requirements that changed as discoveries were made or understanding was increased during development.

In an agile system, these separate tasks may be done in parallel or considered continuous. For example, data may be analyzed multiple times during a project, in different ways and with different objectives, as new needs emerge.

3. Exploring Data

The author has found that having a large number of sample input documents significantly increases the chance a project will be successful. Thousands of tens of thousands of journal articles, for example, selected across every possible journal and journal publisher involved in a project, will likely mean that almost all likely situations will actually occur, and that differences between schemas or DTDs and actual data will be discovered.

Having the *right* schemas, DTDs, entity files, and input data, and the *right* documentation is essential.

If there is sample output, it should be compared to what is generated, so having corresponding input and output is a major help.

If you are faced with exploring, say, five thousand documents, the first thing is to validate them. Clearly you don’t want to lead each document by turn into an XML editor and press Validate. But a simple XSLT stylesheet might work:

```
<xsl:template match="/">
      <xsl:for-each select="uri-collection('data/*.xml')">
         <xsl:try select="doc(.)">
           <xsl:catch>{.} failed</xsl:catch>
         </xsl:try>
      </xsl:for-each>
    </xsl:template>
```

Exact details will vary depending on how the XSLT implementation handles collections; an alternative is a simple bash script that writes a list of files, and using unparsed-text-lines() to read a URI at a time, instead of using collection(), perhaps like this:

```
ls data > file-list.txt
```

or more, if escaping special characters such as & is needed.

If validating is too slow, you could use xmllint instead of Saxon on most platforms (you might need to install it). Since modern computers support multiple programs running at the same time independently, you could validate groups of sample files in parallel, again maybe with a script.

When you write small scripts in this way, make sure to put a comment at the start of each to say how to use them, and make a Makefile or a runme.sh file that runs them, to make it easy to return to this task later.

Validating files usually involves setting up an XML catalog; you might find it helpful to use **strace** as a wrapper to see which XML catalog files are being opened.

If you do this sort of basic exploration often, consider writing a simple tool. A half hour spent working on it will more than repay itself on the second project, or later in the first one. It could be a simple shell script and take options for whether to trace files opened, and in which directory (folder) to look for data files, and which catalog to use.

This level of scripting is very easy and highly productive. Be *very* careful to remember the comments, though:

```
#! /bin/sh
  # validate all files in the data directory
  # options: -trace - turn on catalog file tracing
  TRACE=
  if test "$1" = "-trace"
  then
      TRACE="strace -e openat,open"
      shift; # delete the -trace option
  fi
  $TRACE xml-validate "$@"
```

There is no need to use 1960s big-business style comments with dates and who wrote what. If necessary, use a git repository, for example on gitlab.com, and store all your scripts there, and git will tell you who added each line and when, if you need to know.

Writing simple tools like this will increase your confidence and skills. Even if you are an experienced programmer with decades of scripting experience, the exercise will get you into the right head-space for working in the project. Think of it as like doing lunge exercises before a race!

4. Off-the-shelf tools and ad-hoc tools

After sending some time figuring out which schemas or DTDs to use (and it is usually not worth worrying too much about getting them all right at first: the client might not have supplied all of the right files or they might do database imports without validation), it's time to look inside, to look at the data.

A helpful initial question is, “what elements are used, and how often?” We could answer this for example in XPath or XQuery:

```
distinct-values(//name())
```

You might need to use something like db:open('data')//name(), depending on your XQuery or XPath processor.

The resulting list is unsorted, and doesn't tell us how often each element name occurred.

```
//name() => distinct-values() => sort() => string-join('&#x10;')
```

might give us a sorted list. We would like the numbers, too:

```
for $n in (//name() => distinct-values() => sort())
return $n || ' ' || count(//*[name() eq $n])
```

This gives ugly output like this:

```
0
a 775223
```

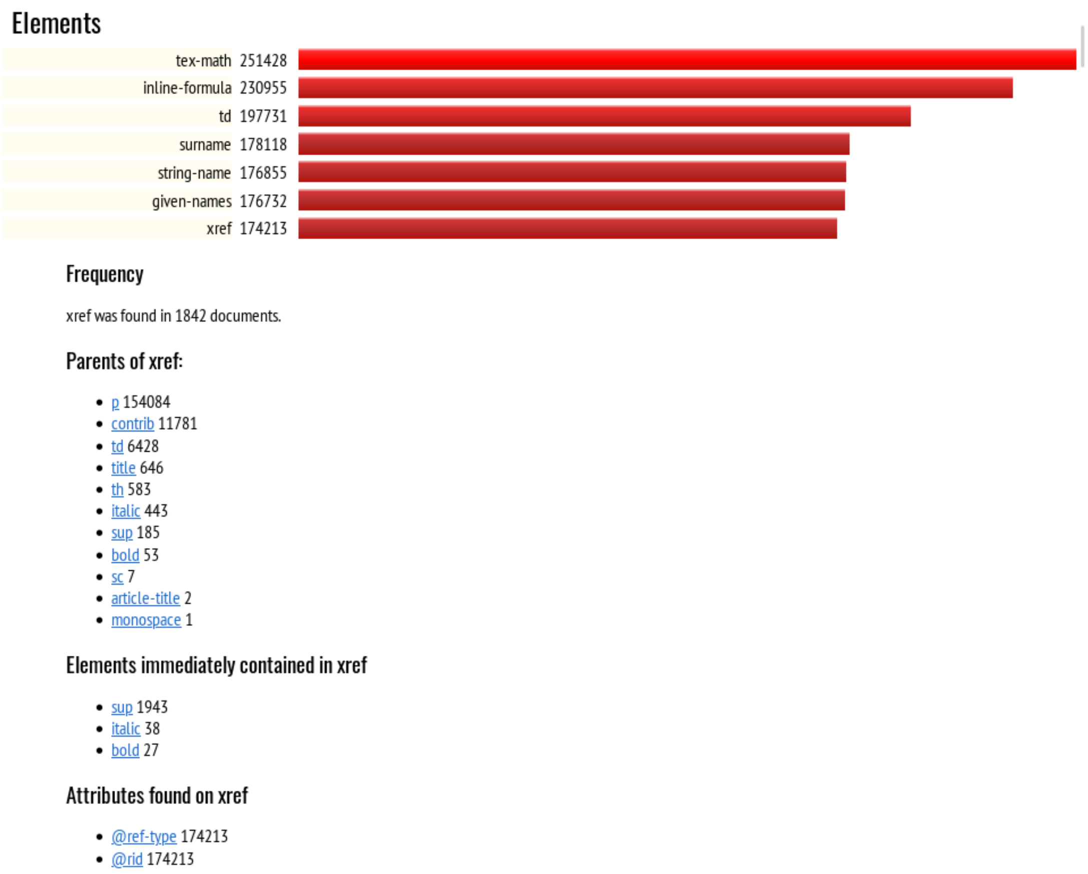
```
body 9669
br 9608
circle 8718
div 265564
font 19208
form 19232
  . . .
```

The empty name with a zero is from something without a name: the document node. We can use //*/name() instead and it will go away.

The output is still ugly. We could use some formatting, or maybe generate HTML. And after that, we might want the same for attributes. Then maybe we want to know about attribute values. With thousands of sample documents, though, there are a lot of values perhaps for ID and for dates that we might want to ignore.

You can see where this is going. Put all this into a script and comment it, give it some options or stylesheet parameters, and you can also run it on the output you generate and account for the numbers. Figure Figure 1 [181] shows what a simple HTML report might look like:

Figure 1. FreqX Report for Element Counts



Elements

tex-math 251428
inline-formula 230955
td 197731
surname 178118
string-name 176855
given-names 176732
xref 174213

Frequency

xref was found in 1842 documents.

Parents of xref:

- p 154084
- contrib 11781
- td 6428
- title 646
- th 583
- italic 443
- sup 185
- bold 53
- sc 7
- article-title 2
- monospace 1

Elements immediately contained in xref

- sup 1943
- italic 38
- bold 27

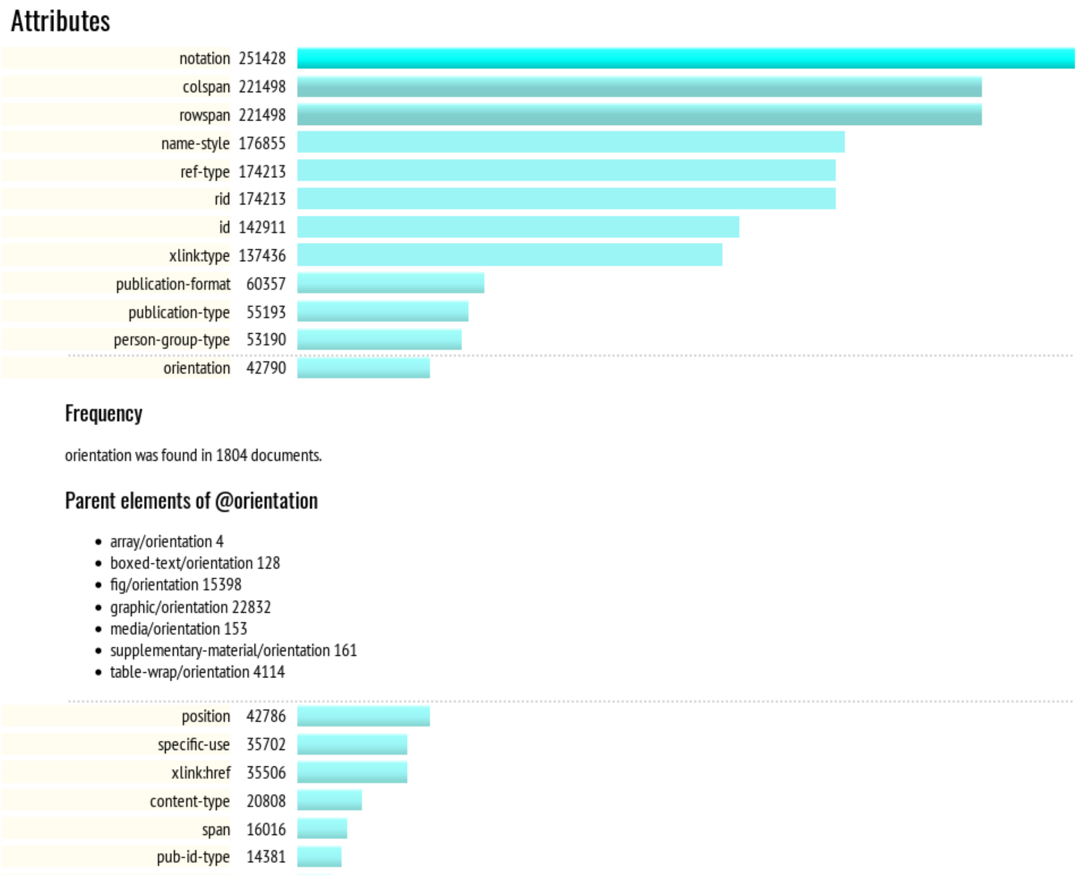
Attributes found on xref

- @ref-type 174213
- @rid 174213

At this point, an ad-hoc query has turned into a tool. Attributes were added, as per Figure 2 [182], which shows the start of the bar chart for elements sorted by frequency, once a user has clicked on, and expanded, the entry for xref. Figure Figure 3 [183] shows the attribute value table, where the user has expanded orientation to get more information.

The FreqX tool is freely available from gitlab; development was funded by Mulberry Technologies. However, it's simple enough that developing a tool to meet your own needs might be easy, as might editing the FreqX source code. If you make changes or fix bugs, please do file issues or patches, so that everyone else can benefit. For example, an often-requested feature is being able to go from an attribute value to a list of documents in which it appears; this might be best accomplished using the eXist-db or BaseX RestXQ feature.

Figure 2. FreqX Report for Attributes



Attributes

notation	251428
colspan	221498
rowspan	221498
name-style	176855
ref-type	174213
rid	174213
id	142911
xlink:type	137436
publication-format	60357
publication-type	55193
person-group-type	53190
orientation	42790

Frequency

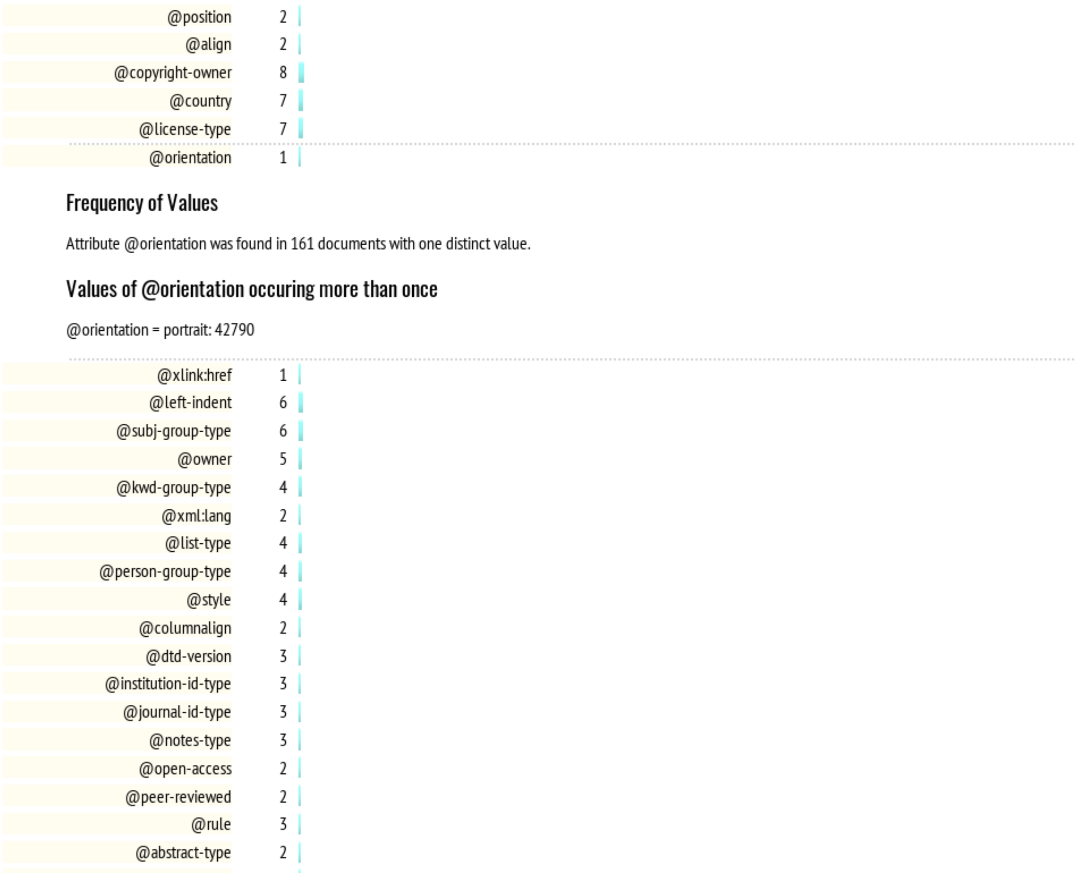
orientation was found in 1804 documents.

Parent elements of @orientation

- array/orientation 4
- boxed-text/orientation 128
- fig/orientation 15398
- graphic/orientation 22832
- media/orientation 153
- supplementary-material/orientation 161
- table-wrap/orientation 4114

position	42786
specific-use	35702
xlink:href	35506
content-type	20808
span	16016
pub-id-type	14381

Figure 3. FreqX Report for Attribute Values



@position	2
@align	2
@copyright-owner	8
@country	7
@license-type	7
@orientation	1

Frequency of Values

Attribute @orientation was found in 161 documents with one distinct value.

Values of @orientation occuring more than once

@orientation = portrait: 42790

@xlink:href	1
@left-indent	6
@subj-group-type	6
@owner	5
@kwd-group-type	4
@xml:lang	2
@list-type	4
@person-group-type	4
@style	4
@columnalign	2
@dtd-version	3
@institution-id-type	3
@journal-id-type	3
@notes-type	3
@open-access	2
@peer-reviewed	2
@rule	3
@abstract-type	2

5. Analyzing Existing Code

One of the difficulties of receiving a bunch of XSLT stylesheets or XQuery files is working out what calls what. Worse, if the files contain errors or might be unfinished, or might have pieces commented out, it can be hard to process them with XML tools. Another difficulty is working out where any particular function or template is defined.

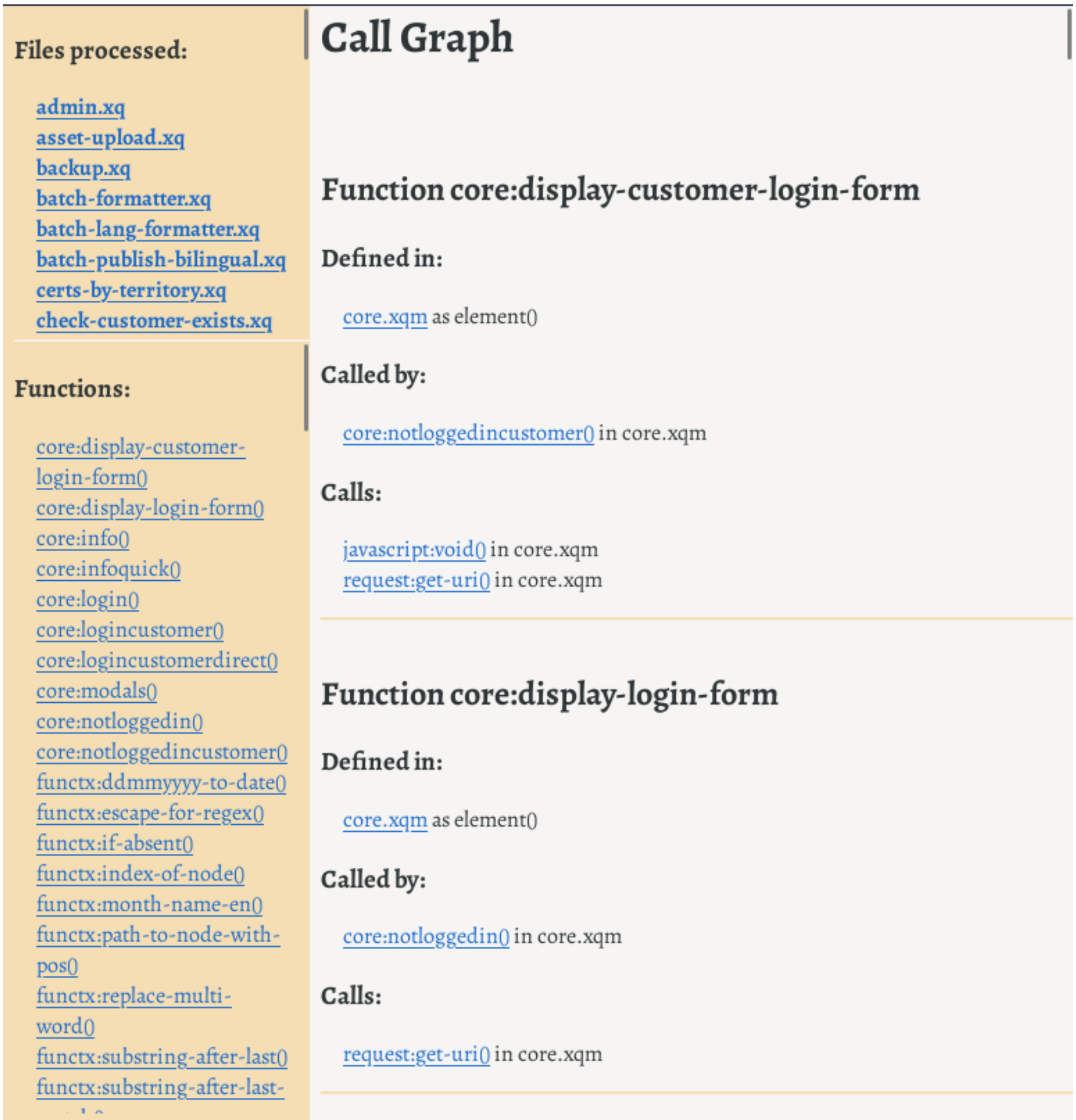
Xarcissus is a tool in two parts. the first is a scanner written in Perl, that tries to do the best it can even when the input isn't well-formed XML. The second is an XSLT stylesheet that takes the output of the scanner and makes a summary, in HTML. An earlier version also drew a dependency graph using the GraphViz library, but this turned out not to be useful in large projects, where the resulting diagram was too complex, and not to be needed in small projects.

Although Xarcissus is not currently distributed, it is freely available from the author on request. The reason for the restriction is that it's really an internal Delightful Computing tool that gets hacked around based on what's needed at any given time, rather than made into a product. For example, it can also produce Swagger (OpenAPI) files from comments in XQuery files.

Figure Figure 4 [184] shows output from Xarcissus on a moderately large XQuery project. It has found XQuery files that were referred to in HTML, in JavaScript, in XSLT, and in XQuery files. This run did not include any XSLT templates, but where those are found, their names and match patterns are recorded[11].

[1] XSLT support is incomplete at the time of writing; it seems to come and go from time to time.

Figure 4. FreqX Report for Attribute Values



Files processed:

- admin.xq
- asset-upload.xq
- backup.xq
- batch-formatter.xq
- batch-lang-formatter.xq
- batch-publish-bilingual.xq
- certs-by-territory.xq
- check-customer-exists.xq

Functions:

- core:display-customer-login-form()
- core:display-login-form()
- core:info()
- core:infoquick()
- core:login()
- core:logincustomer()
- core:logincustomerdirect()
- core:modals()
- core:notloggedin()
- core:notloggedincustomer()
- functx:ddmmyyyy-to-date()
- functx:escape-for-regex()
- functx:if-absent()
- functx:index-of-node()
- functx:month-name-en()
- functx:path-to-node-with-pos()
- functx:replace-multi-word()
- functx:substring-after-last()
- functx:substring-after-last-

Call Graph

Function core:display-customer-login-form

Defined in:

core.xqm as element()

Called by:

core:notloggedincustomer() in core.xqm

Calls:

javascript:void() in core.xqm
request:get-uri() in core.xqm

Function core:display-login-form

Defined in:

core.xqm as element()

Called by:

core:notloggedin() in core.xqm

Calls:

request:get-uri() in core.xqm

The point of Xarcissus is really that you can fairly easily write a tool that is useful. You can use HTML for a simple user interface, and get something working in maybe ten minutes to an hour that will save days. Since the point of this tool is understanding, save the report it makes, maybe also save a copy of the tool with the project, and then modify the tool for the next project. Or make it into a product, with documentation!

6. Doing The Actual Work

Although Eddie 2 has been described elsewhere in some detail, the point of this paper is to describe the philosophy of making tools, so we will take a different approach, and describe its evolution.

In the first instance, the author needed to identify structural differences between two versions of the JATS DTD. This is a fairly large vocabulary, and two organizations that needed to interchange documents each had made their own variation on it.

The first version of Eddie 2 used a DTD parser module in Perl; the XML support in Python at the time did not seem to report DTD events such as finding element or entity declarations. The first version took a few hours to write and get working satisfactorily.

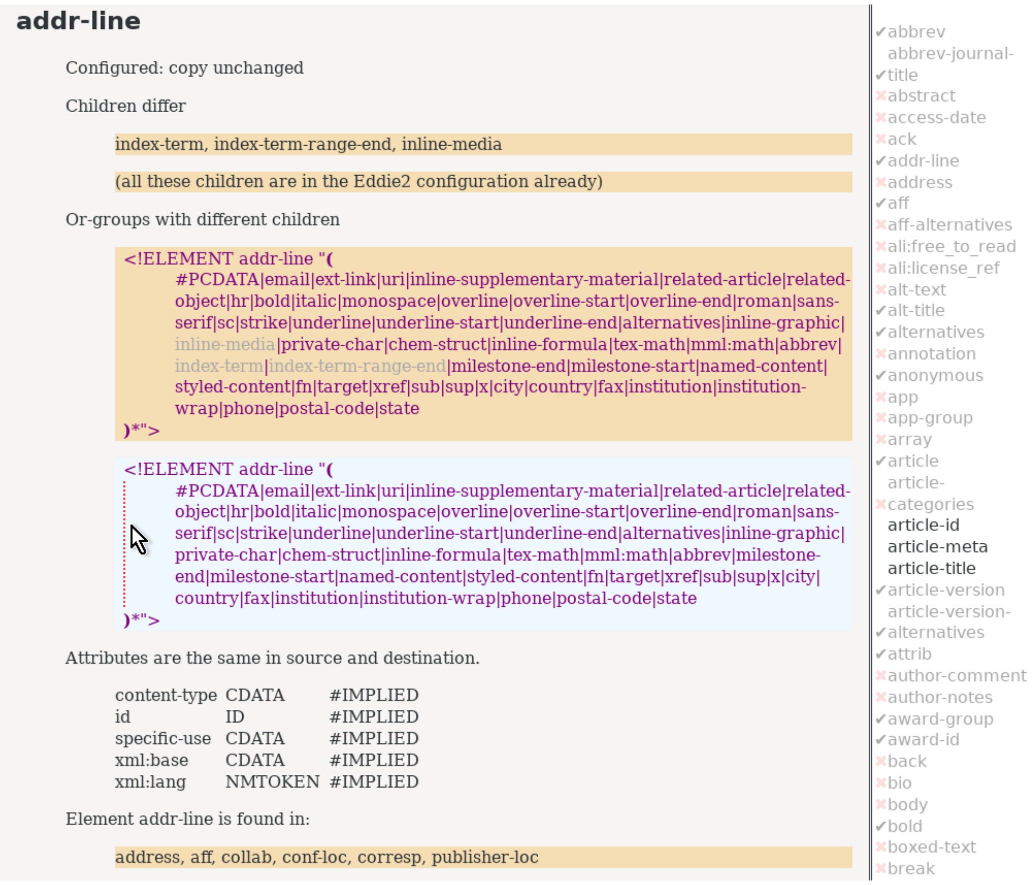
Once Eddie 2 was producing useful output, and could handle parameter entities more helpfully than other dtd-diff tools, the next step was to generate an XSLT stylesheet to handle each element that might appear in the input, simply copying it to its output and producing a message that it had been seen.

Including this XSLT and processing the sample documents available meant that the most frequently-occurring elements could be handled first; this meant the number of validation errors in the output reduced very quickly.

Since Eddie 2 made an HTML report, it was easy to add to this a list of elements that were in a configuration file Eddie 2 read, and to highlight whether the elements had the same content model and attributes in both DTDs, or whether they differed.

This list, shown on the right-hand side in Figure Figure 5 [185], can also be used as a sort of to-do list: any elements that differ between DTDs and are not handled in the XSLT or in the configuration file are marked with a red X. Eddie 2 also reads the XSLT file to check for coverage, and warns if there are such elements with no template to match them in the main stylesheet.

Figure 5. Eddie 2 Report



addr-line

Configured: copy unchanged

Children differ

index-term, index-term-range-end, inline-media

(all these children are in the Eddie2 configuration already)

Or-groups with different children

```
<!ELEMENT addr-line "(
        #PCDATA|email|ext-link|uri|inline-supplementary-material|related-article|related-
        object|hr|bold|italic|monospace|overline|overline-start|overline-end|roman|sans-
        serif|sc|strike|underline|underline-start|underline-end|alternatives|inline-graphic|
        inline-media|private-char|chem-struct|inline-formula|tex-math|mml:math|abbrev|
        index-term|index-term-range-end|milestone-end|milestone-start|named-content|
        styled-content|fn|target|xref|sub|sup|x|city|country|fax|institution|institution-
        wrap|phone|postal-code|state
)*">
```

```
<!ELEMENT addr-line "(
        #PCDATA|email|ext-link|uri|inline-supplementary-material|related-article|related-
        object|hr|bold|italic|monospace|overline|overline-start|overline-end|roman|sans-
        serif|sc|strike|underline|underline-start|underline-end|alternatives|inline-graphic|
        private-char|chem-struct|inline-formula|tex-math|mml:math|abbrev|milestone-
        end|milestone-start|named-content|styled-content|fn|target|xref|sub|sup|x|city|
        country|fax|institution|institution-wrap|phone|postal-code|state
)*">
```

Attributes are the same in source and destination.

```
content-type CDATA    #IMPLIED
id           ID       #IMPLIED
specific-use CDATA    #IMPLIED
xml:base     CDATA    #IMPLIED
xml:lang     NMTOKEN  #IMPLIED
```

Element addr-line is found in:

address, aff, collab, conf-loc, corresp, publisher-loc

✓abbrev
abbrev-journal-
✓title
✗abstract
✗access-date
✗ack
✓addr-line
✗address
✓aff
✗aff-alternatives
✗ali:free_to_read
✗ali:license_ref
✗alt-text
✓alt-title
✓alternatives
✗annotation
✓anonymous
✗app
✗app-group
✗array
✓article
article-
✗categories
article-id
article-meta
article-title
✓article-version
article-version-
✓alternatives
✓attrib
✗author-comment
✗author-notes
✓award-group
✓award-id
✗back
✗bio
✗body
✓bold
✗boxed-text
✗break

The Eddie 2 tool has proven to be highly effective in helping people write this sort of transformation between similar DTDs. Because it depends on some Perl modules, it can be a little tricky to install; a replacement written entirely in XSLT is waiting for a project to come along and fund its completion.

7. Conclusion

Writing tools is easier than it sounds, and is satisfying. You quickly build up tools that you can use, and, rather like a woodworker with a feather-board for pushing material through a table saw, the cost can be low and the benefit very high.

The tools mentioned here are (or may be) available either from gitlab.com/barefootliam or directly from the author.

Markup UK