XML Prague 2026

Conference Proceedings

Prague University of Economics and Business
Prague, Czech Republic

June 4–6, 2026

XML Prague 2026 – Conference Proceedings

ISBN 978-80-907787-4-0 (pdf)
ISBN 978-80-907787-5-7 (ePub)

Table of Contents

General Information

Date

June 4th, 5th and 6th, 2026

Location

Prague University of Economics and Business
W. Churchill Sq. 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Petr Cimprich, *WPP & XML Prague, z.s.*
Vít Janota, *XML Prague, z.s.*
Káťa Prouzová, *XML Prague, z.s.*
Jirka Kosek, *xmlguru.cz & XML Prague, z.s.*
Martin Svárovský, *XML Prague, z.s.*
Mohamed Zergaoui, *Optimizix*

Program Committee

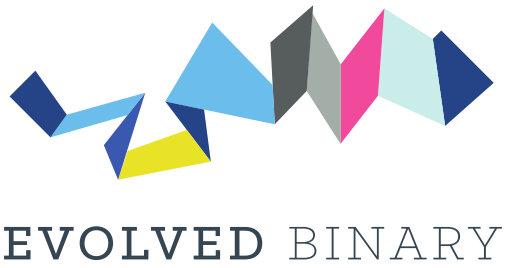
Petr Cimprich, *WPP*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *Prague University of Economics and Business*
Ari Nordström, *Creative Words*
Jennifer Ramirez-Betancur, *ZB MED*
Adam Retter, *Evolved Binary*
Andrew Sales, *Andrew Sales Digital Publishing*
Felix Sasaki, *SAP SE*
John Snelson, *Progress Software*
Sheila Thomson, *Saxonica*
Norman Tovey-Walsh, *Saxonica*
Eric van der Vlist, *Dyomedea*
Priscilla Walmsley, *Datypic*
Lauren Wood, *Textuality, xml.com*
Mohamed Zergaoui, *Optimizix*

Produced By

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

Sponsors

oXygen (https://www.oxygenxml.com)
Antenna House (https://www.antennahouse.com/)
le-tex publishing services (https://www.le-tex.de/en/)
Saxonica (https://www.saxonica.com/)
Evolved Binary (https://evolvedbinary.com/)



Preface

This publication contains papers presented during the XML Prague 2026 conference.

In its 18th year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup, extensible web, publishing and digital books, XML technologies for big data, utilization of AI in XML technologies and more. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 4–6 June 2026 at the campus of Prague University of Economics and Business. XML Prague 2026 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, Prague University of Economics and Business.

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

The Thursday run in an un-conference style which provides space for various XML community meetings in parallel tracks. Friday and Saturday is devoted to a classical single-track format and papers from these days are published in the proceeedings.

Since 2022 both Markup UK and XML Prague conferences are held in alternate years. We are looking forward to meeting you in May/June 2027 in London and in June 2028 back in Prague.

We hope that you enjoy XML Prague 2026!

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

Invisible XML: State of Play and Future Directions

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Abstract

Invisible XML [6] has had a stable specification since 2022, there are currently a half dozen implementations, and typically a dozen presentations per year have recently been given at conferences. At the beginning of 2026 the first International Symposium on Invisible XML was held [17], with 14 presentations and 40 or so attendees. Meanwhile there is a working group [21] developing the language further.

This paper gives an overview and discussion of the topics and issues currently being considered within the community on the route to the next version.

Keywords: markup, invisible xml, ixml, notation design, parsing, standards

1. Introduction

Since its introduction, ixml usage has been increasing over a surprisingly broad range of applications, including art [2], analysing late Roman trials [18], aircraft maintenance parts ordering communications [11], designing knitting patterns [7], processing Vehicle Identity Numbers [20], banking [3], parsing literature [8], and many others. With this experience emerges requests for features or support for use cases. These requests get noticed by, or passed on to, the working group for consideration, and often arrive as requests for particular features, for instance, similar features available in other languages.

There are two observable streams in notation design, the one which could be characterised as 'kitchen sink' design, adding all imagined features to the language without consideration of how they fit together; the programming language *perl* can be seen as an example of such a design. The other stream is about recognising requirements, and finding unified, consistent ways of satisfying those requirements, *python* being a contender for being an example of such a design: see for instance *The Zen of Python* [24] for a set of rules that was used to direct the design of the language. Einstein's maxim "Everything should be kept as simple as possible, but no simpler" is a good aim when designing: it implies not adding two

different mechanisms to achieve the same result, which means designing from use cases, and not features.

The rest of this paper will discuss individual issues under consideration within the community.

2. Renaming during Serialisation

Rules in ixml define two things: 1) the input syntax, and 2) the details of how the resulatant parsetree should be serialised. In ixml 1.0 the serialisation of an element or attribute *always* has the same name as the rule it comes from.

However, two different input syntaxes might represent the same output serialisation (for instance two different date formats). This originally meant that one abstraction, with two different input formats, had to have two separate output formats as well.

Renaming gives you more control over this, by allowing you to use a different name from the rule name on serialisation. Without renaming, you get

```
date: y, m, d.
```

⇒

```
<date><y>2025</y><m>08</m><d>06</d></date>
```

While using renaming gives a different serialisation:

```
date > d: y, m, d.
```

⇒

```
<d><y>2025</y><m>08</m><d>06</d></d>
```

This is completed work that is already agreed on by the group, but not yet officially published. It gives control over the element and attribute names used in serialisation and is already widely implemented. See the modularisation paper [10] for examples of its use, and the work-in-progress ixml draft [5] for its definition.

3. Modularisation

Since the release of ixml, two developments have been observed: firstly, some very large grammars are emerging, such as the one for XPath 4 [23] at 350 lines, and 200 rules; the second is that several grammars are being written with common subparts, for instance for URIs, and indeed XPath expressions.

Modularisation addresses these issues by allowing you to split grammars into smaller more manageable parts, and allowing you to specify which rules may be used by other grammars, while at the same time protecting you from name clashes across the combined grammar. For example here is the proposed syntax of a module, using its own definition:

```
+uses ixml, name, s, RS from ixml.ixml;
iri from iri.ixml
+shares module

module: s, (multiuse; shares)*, ixml.
-multiuse: -"+uses", RS, uses++(-";", s).
shares: -"+shares", RS, entries.
uses: entries, RS, -"from", RS, from.
-entries: share++(-",", s).
share: @name, s.
@from: iri, s.
```

The proposal allows modularisation to be done by a preprocessor, so that a processed modularised grammar can still be fed to implementations that don't support modularisation directly.

See the paper from MarkupUK 2025 [10] and a slightly different proposal from Norm Tovey-Walsh [9].

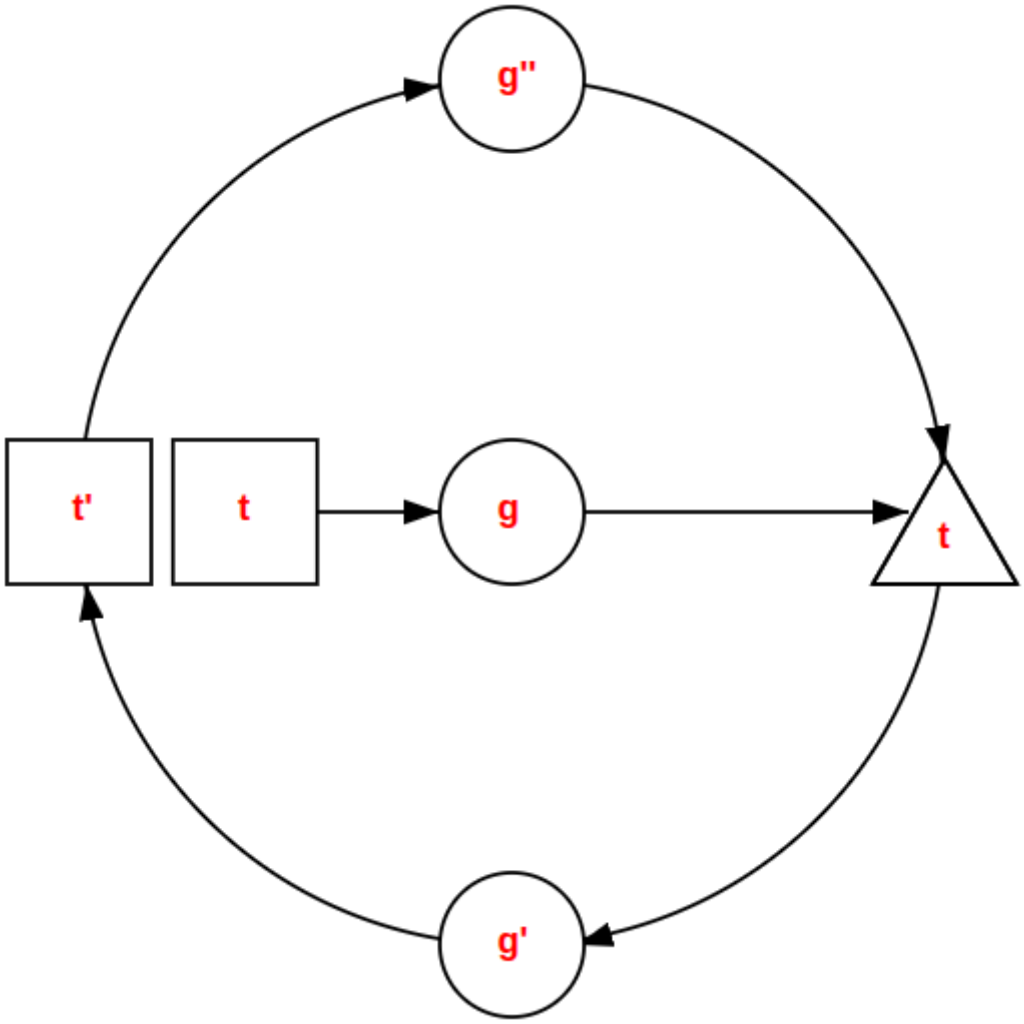
4. Round tripping

Round tripping is about recreating an equivalent input document from the output of ixml.

At first sight, it would appear that round tripping of ixml, recreating the identical document that produced the output, is not in general possible, since an ixml grammar can delete characters that were in the input, and add other characters to the output. However, by defining round tripping as "producing a document that would produce the same output", it is possible in the general case.

The process works by transforming the ixml grammar that produced the output into one that recognises the output produced by the original grammar, and serialising only the terminal characters of a resultant parse, thus producing a textual document that when processed with the original grammar produces the same output. The one implementation to date [15] refers to this process as "creating the canonical value", since although the new document is not character-for-character identical with the original, running it through ixml, and back again will continually produce the same textual document.

An interesting corollary to this is that transforming the transformed grammar in the same way a second time produces the same output as the original grammar, but requiring less work from the ixml serialiser, pointing to a possible way of simplifying the ixml serialiser.



The round tripping process. Input t is the original input, which is processed using the original grammar g to produce output t. Transforming grammar g to produce grammar g' allows the output t to be transformed back to input document t', which though not identical to t, will produce the same output as t when processed using grammar g. However, transforming g' using the same transformation process to create grammar g'' will also produce the identical output, but with far simpler use of the ixml serialiser.

This is further described in [16].

5. Ambiguity

Ambiguity in ixml is a property of the input.

Although the aim of ixml is to describe input formats so that their structure can be recognised without the need for markup, ambiguity is nevertheless deliberately accepted, firstly from a usability point of view, since it allows you to get some result, with a warning that there are other possibilities, and secondly

because some *input* ambiguities may not produce an *output* ambiguity, in which case it makes no difference which parse gets serialised.

Since ixml does not dictate which parser should be used, but only the properties of an acceptable parser, it is not reasonable to require a particular parse of the ambiguous input, so it is unspecified which parse is serialised; this means that different implementations may produce a different result.

Although ixml permits and handles ambiguity, there are some disadvantages to having an ambiguous grammar: apart from the uncertainty of whether the serialisation represents what the grammar writer intended the structure of the document to be, there is the disadvantage of speed, since an ambiguous grammar requires the input to be parsed multiple times.

5.1. Types of ambiguity

Ambiguities can be divided into several classes.

Some ambiguities are only on the input, such as deleted spaces in this example:

```
input: number*.
number: spaces, digit, spaces.
digit: ["0"-"9"].
-spaces: -" "+.
```

Even though this is ambiguous, all parses will produce an identical serialisation.

Some ambiguities are due to badly written grammars. For instance,

```
expression: number; identifier; expression, op, expression.
op: ["+-×÷"].
identifier: [L]+.
number: ["0"-"9"]+.
```

For an input like

```
a-b-c
```

this will produce two different parses one effectively

```
(a-b)-c
```

and the other

```
a-(b-c)
```

which have different meanings. The grammar has been incorrectly written.

Nothing should be done about this case. There is no doubt that tracking down ambiguities can be difficult, and time-consuming for inexperienced grammar writers, but trying to solve this by *ad hoc* means doesn't solve the underlying problem that the input is not properly described, nor would it guarantee that you

get the serialisation that you want in all cases. It is a potential source of technical debt.

Some ambiguities are inherent in the input, such as US/World dates, like `4/5/2026`,

```
date: us; world.
us: month, "/", day, "/", year.
world: day, "/", month, "/", year.
day: d, d?.
month: d, d?.
year: d, d, d, d.
d: ["0"-"9"].
```

Still, even in these cases it is possible to rewrite it to an unambiguous *grammar* that explicitely identifies ambiguous and non-ambiguous *cases*:

```
date: us; world; ambig.
us: month, "/", day, "/", year.
world: day, "/", month, "/", year.
ambig: md, "/", md, "/", year.
day: "1", ["3"-"9"]; "2", "0"-"9"; "3", ["0"-"1"]. {13-31}
month: "0"? ["1"-"9"]; "1", ["0"-"2"].              {1-12}
md: -month.
```

Although this still identifies dates like `1/1/2026` as ambiguous, since they are syntactically ambiguous, even though they aren't semantically ambiguous, even these cases are in principle handleable with a yet more complex grammar; this wouldn't disambiguate the cases though, just choose one in favour of the other.

5.2. Approaches

There are two possible approaches to dealing with ambiguity. One is to allow it, and add the ability to select amongst ambiguous parses. The other is to add expressiveness to the grammar notation to ease making unambiguous grammars.

For the first, some approaches add priorities to rules [12], [14], that allow the selection of one rule above another when it comes to ambiguity. There are some dangers to this approach; for instance CSS [4] has a similar feature where the `!important` keyword gives a rule priority over others. This was introduced for a very particular (legal) use case, but has proven to be an enormous source of technical debt, since people often tend to use it for a quick fix, making stylesheets that use it fragile, and hard to update.

One of the observable problems of specifying grammars is the difficulty of splitting input into distinct cases. For example:

```
catalogue: entry*.
entry: header, item+.
header: text, code, #a.
```

```
item: text, #a.
text: word++" ".
-word: (l; d)+.
-l: [L].
-d: ["0"-"9"].
@code: l, l, l, d, d, d.
```

Example input:

```
Fiction fic001
Ulysses
Brave New World
1984
NonFiction non123
Translating Beaudelaire
The Sixth Extinction
```

The underlying problem here is a badly designed input format, but it is necessary to deal with the real world and handle formats like this. In this case, `word` and `code` are not distinct, so `header` also matches `item`. You could add a priority to a rule:

```
header: text, code, #a. !important
```

which would say that in the case of ambiguity, choose this rule, but this doesn't explicitly express what is going on, making it hard to understand, and harder to update later, nor does it take away the problem of having to parse an ambiguous document in the first place.

A better solution would be the ability to say explicitly "An item is any string of characters where the last word doesn't match a code." For instance,

```
item: word**" ", " ", lastword, #a.
-lastword: word!code.
```

Here `word!code` means a word, as long as it doesn't match a code. This approach will mentioned more shortly.

6. Spaces

In passing it is worth mentioning the problem of spaces, since they are a typical source of ambiguity, and one of the harder aspects of free-format inputs for inexperienced grammar writers. This may be because classically there is a lexical analyser feeding tokens to the parser, so that the parser never sees spaces, and so the grammar doesn't need to mention them.

One way to deal with this in ixml is to have a 'lexical' part to the grammar, that simulates the lexical analyser, and deals with spaces between symbols:

```
OPEN: "(", s.
CLOSE: ")", s.
PLUS: "+", s.
```

and so on, so that spaces don't have to appear in the main body of the grammar. Adding a lexical analyser to ixml wouldn't solve this problem, because the syntax of tokens still have to be described, and ixml already has a method of describing syntax.

Since spaces are such a recurring problem, it would be tempting to define a mode of processing where if an input character fails to match, and it turns out to be a space, then it is just ignored, but in truth, there are few input languages where spaces are never relevant. In the 60's programming languages like FORTRAN and the Algols were explicitly designed so that spaces were never relevant (even in strings, in the case of the Algols), but nowadays that is seldom the case. For instance in CSS `p.note` and `p .note` have very different meanings. Even ixml has places where spaces are required.

7. Lexerless parsing

As mentioned, traditionally parsing has been done in two stages, with two parsers running in parallel, a low-level lexical analyser, whose input is a string of characters, and whose output is a string of 'tokens', and then the main parser whose input is those tokens, and the output a parse tree.

This was needed traditionally to enable the use of non-general parsing algorithms such as LL(1), which would otherwise not be possible in most cases. However, there are new approaches that add constructs to the syntax description method that remove the need for a lexical stage. See for instance [19].

The current ixml proposal is to add one construct, with the syntax not yet finalised. Alongside A*, A+, A?, a construct A! is added to mean "An A may not appear here".

For example,

```
identifier: letter+, letter!.
```

means "The longest stretch of letters that can be matched". Thus, with

```
keyword: "if", letter!;
"then", letter!;
"else", letter!.
identifier: keyword!, letter+, letter!.
```

this last rule would then mean "an identifier is the longest string of letters that is not a keyword".

8. Namespaces

In designing XML, the group responsible did a clever thing when adding a notation for namespaces [22]: they designed the namespace declarations to look like attributes, so that XML documents would be syntactically compatible with earlier software, but they would have a different semantic interpretation because they begin with the characters `xmlns`.

The same approach could be used for ixml: by specifying that things that look like attributes in the serialisation but begin with the characters `xmlns` should be interpreted as namespace declarations. For implementations that produce textual output, this adds no extra processing; for implementations that go directly to an XML internal form, the namespace declarations have to be recognised and handled appropriately, as they are in XML processors.

Accepting this, you could define a rule whose output is the shell of an HTML document to include a namespace in this way:

```
html: xhtml-ns, head, body.
@xhtml-ns>xmlns: +"http://www.w3.org/1999/xhtml".
```

which would give

```
<html xmlns='http://www.w3.org/1999/xhtml'>
```

9. Greedy Matching

A problem for beginners coming to ixml is that they may have internalised idioms from other similar systems that work differently from ixml. A good example of this is greedy matching as used in typical regular expression recognisers, where in many regular expression implementations the pattern

```
["a"-"z"]*
```

matches the longest-possible stretch of lower case letters; however, in traditional grammar usage it represents *any* length that fits in the context of where it is used, and not necessarily the longest.

An option would be to introduce a separate notation to specify the longest possible stretch, for example

```
["a"-"z"]>>
```

which then for consistency would require a similar construct for separated repeats

```
["a"-"z"]>>(",", s?)
```

However, as already seen, the negation construct would already allow the specification of longest stretches, so it is not clear that adding another construct for this explicit case would be necessary.

10. Numbered Repeats

Grammars in ixml have extra constructs available, not available in traditional grammars, and not mentioned in traditional parsing algorithms, in particular the structures for repeated constructs with separators. As a section in the ixml specification points out, it is easy to handle these constructs, partly thanks to serialisation control, by transforming the grammar into an equivalent one that doesn't use the constructs.

Some grammar systems allow the specification of numbered repeats, for instance "zero or more up to 6 letters". As an example ABNF [1] allows

```
3 digit
```

to specify exactly 3 digits,

```
1*4 digit
```

to specify 1-4 digits, and so on.

There are very few grammars that requires such a notation, though it would be easy to transform a grammar using such a construct into one not using it.

11. Pragmas

This is surprisingly a contentious issue: how to address individual pieces of software.

Software occasionally provides a mechanism for instructing a processor to act in a certain way. XML itself has Processing Instructions, that consist of a *target* that specifies what the pragma is about, and *content*, which is used by software that processes such instructions. For instance

```
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
```

These are typically a type of comment: they don't alter the semantics of the language, but instruct a mode of operation to the processor.

A paper ??? proposed using pragmas not only to address individual software, but also as an extension mechanism. The danger of this is that conflating the two aspects risks undermining the interoperability of ixml.

Since pragmas are there to talk to individual pieces of software, it would seem advisable to let the software specify what they expect, within a broad but simple structure in the style of XML processing instructions: identify the target, and let the target do the rest. A pragma should be identifiable as such, and the content should be up to the addressed software.

12. Versioning

Most software (for instance programming languages) doesn't require its input to specify which version of the processor is required. In fact XML is a bit of an

exception on this point, and it is not obvious what the advantages are to the user of requiring it; it certainly seems to have obstructed adoption of new versions. It may be used as a pragma to the processor to require a certain type of processing or checking, but this is only really necessary when the semantic meaning of a particular syntactic structure has changed between versions.

The current method of specifying the version was added in haste shortly before publication of the specification, which was a mistake, because it left no time to implement and try it out beforehand. In general, a user shouldn't have to be confronted with the need to know which version they are using. As a result, the absence of a version should always be taken to mean "use the most recent version"

13. Conclusion

The adage "Good food takes time" applies equally well to design: there are many interlocking decisions where a change in one part can affect the design of another part, and they need to be designing to achieve a good symbiosis with each other, and then user tested to see the real-life effects of the changes.

The design of the first version of ixml itself went through several iterations, and had small-scale user testing before it ended up as version 1.0. The next version, 1.1, or 2.0, however it will be numbered, has many, sometimes apparently conflicting, requirements that need to be resolved and meshed together.

References

[1] D. Crocker. *RFC 5234 Augmented BNF for Syntax Specifications: ABNF*. ietf.org. 2008. https://datatracker.ietf.org/doc/html/rfc5234 .

[2] Mary Holstege. *Invisible Fish: API Experimentation with InvisibleXML*. In Proceedings of Balisage: The Markup Conference 2024 vol. 29. 2024. https://doi.org/10.4242/BalisageVol29.Holstege01 .

[3] Steven Pemberton. *Banking with ixml and XForms*. Proc. Declarative Amsterdam. 2024. https://declarative.amsterdam/article?doi=da.2024.pemberton.banking .

[4] Håkon Wium Lie et al.. *Cascading Style Sheets level 1*. W3C. 1996. https://www.w3.org/TR/CSS1/ .

[5] Steven Pemberton. *Invisible XML Specification Community Group Editorial Draft*. Invisible XML Organisation. 2026. https://invisiblexml.org/current/ .

[6] Steven Pemberton. *Invisible XML Specification*. Invisible XML Organisation. 2022. https://invisiblexml.org/1.0/ .

[7] Bethan Tovey-Walsh. *When women do algorithms: a semi-generative approach to overlay crochet with iXML and XSLT*. In Proceedings of Balisage: The

Markup Conference vol. 29. 2024. https://doi.org/10.4242/BalisageVol29.Tovey-Walsh01 .

[8] Steven Pemberton. *The Book of Doublends Jined: Parsing Finnegans Wake with ixml*. In Proceedings of Balisage: The Markup Conference vol. 30. 2025. https://doi.org/10.4242/BalisageVol30.Pemberton01 .

[9] Norm Tovey-Walsh. *An Invisible XML modularity proposal*. Nineml. 2026. https://nineml.org/proposals/2026/modularity/ .

[10] Steven Pemberton. *Modular ixml*. Proc. MarkupUK 2025. pp 6-20. https://markupuk.org/pdf/proceedings-2025-2.pdf .

[11] Ari Nordström. *Adventures in Mainframes, Text-based Messaging, and iXML*. In Proceedings of Balisage: The Markup Conference vol. 29. 2024. https://doi.org/10.4242/BalisageVol29.Nordstrom01 .

[12] Bryan Ford. *Parsing Expression Grammars: A Recognition Based Syntactic Foundation*. Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2004. 111–122. 10.1145/964001.964011. 1-58113-729-X.

[13] Tomos Hillman et al.. *Designing for change: Pragmas in Invisible XML as an extensibility mechanism*. In Proceedings of Balisage: The Markup Conference vol. 27. 2022. https://doi.org/10.4242/BalisageVol27.Sperberg-McQueen01 .

[14] E. Shinan. *Lark: A parsing toolkit for python*. github. 2025. https://github.com/lark-parser/lark .

[15] Alain Couthures. *Text normalization with Invisible XML round-tripping*. Proc Declarative Amsterdam. 2025. https://declarative.amsterdam/article?doi=da.2025.couthures.grammix .

[16] Steven Pemberton. *Round-tripping Invisible XML*. in Proc. XML Prague. 2024. 153-164. 978-80-907787-2-6. https://archive.xmlprague.cz/2024/files/xmlprague-2024-proceedings.pdf#page=163 .

[17] ixml org. *The First International Symposium on Invisible XML*. invisiblexml.org. 2026. https://invisiblexml.org/events/symposium2026/ .

[18] C.M. Sperberg-McQueen. *"From Word to XML via iXML: a Word-first XML workflow in the TLRR 2e project"*. In Proceedings of Balisage: The Markup Conference vol. 29. 2024. https://doi.org/10.4242/BalisageVol29.Sperberg-McQueen01 .

[19] M.G.J. van den Brand et al.. *Disambiguation Filters for Scannerless Generalized LR Parsers*. In Lecture Notes in Computer Science vol 2304. 2002. https://doi.org/10.1007/3-540-45937-5_12 . https://cwi.nl/~jurgenv/papers/CC-2002.pdf .

[20] Ari Nordström. *It's Useful After All — VIN Numbers, DITA, and iXML*. Proc XML Prague. 2024. 295-306. https://archive.xmlprague.cz/2024/files/xmlprague-2024-proceedings.pdf#page=305 .

[21] Invisible Markup. *Invisible Markup Community Group*. 2026. W3C. https://www.w3.org/community/ixml/ .

[22] Tim Bray et al.. *Namespaces in XML 1.0*. W3C. 2009. https://www.w3.org/TR/xml-names/ .

[23] John Lumley. *Invisible XML workbench*. Github. 2024. https://johnlumley.github.io/jwiXML.xhtml .

[24] Tim Peters. *PEP 20 - The Zen of Python*. python.org. 2004. https://peps.python.org/pep-0020/ .

Crane-txt2xml – an attempt to socialize XML for non-XML'ers

G. Ken Holman
<gkholman@CraneSoftwrights.com>

Keywords: iXML, PubMed, XML, XSD, OLSPub, OASIS UBL

1. Introduction

I participate in two communities of non-XML-aware users who suffer the imposition of having to use XML syntax for their work. Almost as if the users are allergic to angle brackets. And surely these are not the only two communities where such non-XML'ers can be found.

Communities author information in XML content in accordance with a document model or schema. These models are expressed using constraint semantics for the naming and nesting of portions of content. While one can readily label the start of portions of content, one critical XML angle bracket challenge to eliminate (or at least reduce) is the need to properly close off portions of content with end tags.

The domain of information, such as business documents or biomedical research metadata for only two examples, is not really important to the problem. The problem is that subject matter experts in business or research are not XML syntax experts. Yet, there are situations where computers are not creating such documents and these people have the need to create or edit XML syntax thrust upon them.

Not all XML vocabularies have editing environments that their communities can use to hide the vagaries of XML syntax. And they can be a challenge to use a simple text editor for XML angle brackets, with after-the-fact schema validation the only means by which their use of syntax is checked. Moreover, the validation reports tend not to be geared to non-XML'ers, but this is understandable.

The Crane-txt2xml[1] environment is my attempt at socializing XML for non-XML'ers by providing in multiple domains a simple text-based syntax that gets processed into richly structured XML according to a document model schema. Though a caveat is that not all document model schemas can be supported in this environment that makes some basic assumptions.

The <PubNote>[2] environment includes a real-world configuration of this reusable Crane-txt2xml environment. It supports small biomedical research pub-

[1] github.com/CraneSoftwrights/Crane-txt2xml?tab=readme-ov-file#readme
[2] https://github.com/realtaonline/pubnote?tab=readme-ov-file#readme

lishers in the creation of schema-valid XML for input to OLSPub and PubMed databases.

The mantra for editing, simply, is to label content values in the correct order (officially "document order" as defined by XML). When something is detected out of order, the process stops and the operator is guided to the point in the input that needs to be addressed. If everything is edited in the correct order, then a post process is used to validate the character content found inside the ordered elements.

2. Simple syntax for non-XML'ers

As an introduction to the text syntax used to covert to XML, consider a simple XSD schema to describe a recipe:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="AmountType">
    <xs:simpleContent>
      <xs:extension base="xs:decimal">
        <xs:attribute name="unit"        type="xs:string"
use="required"/>
        <xs:attribute name="approximate" type="xs:string"
use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

  <xs:complexType name="IngredientType">
    <xs:sequence>
      <xs:element ref="Name"/>
      <xs:element ref="Amount"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="RecipeType">
    <xs:sequence>
      <xs:element ref="Title"/>
      <xs:element ref="Ingredient" maxOccurs="unbounded"/>
      <xs:element ref="Step"       maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Title"      type="xs:string"/>
  <xs:element name="Name"       type="xs:string"/>
  <xs:element name="Step"       type="xs:string"/>
```

```
  <xs:element name="Amount"     type="AmountType"/>
  <xs:element name="Ingredient" type="IngredientType"/>
  <xs:element name="Recipe"     type="RecipeType"/>

</xs:schema>
```

A sample XML instance that the user would like to make might be:

```
<?xml version="1.0" encoding="UTF-8"?>
<Recipe>
   <Title>Pancakes</Title>
   <Ingredient>
      <Name>Flour</Name>
      <Amount unit="cups">2</Amount>
   </Ingredient>
   <Ingredient>
      <Name>Maple Syrup</Name>
      <Amount unit="tablespoon" approximate="yes">3</Amount>
   </Ingredient>
   <Step>Mix ingredients together</Step>
   <Step>Cook on a greased griddle □</Step>
   <Step>Serve</Step>
</Recipe>
```

The objective is to have as few rules as possible for the author to remember. I started off the syntax design with a simple use of heralds to label the information, and escape sequences including the entry of Unicode:

```
Recipe:
  Title: Pancakes
  Ingredient:
    Name: Flour
    Amount: @unit:cups 2
  Ingredient:
    Name:"Maple Syrup"
    Amount: @unit:tablespoon @approximate:yes "3"
  Step: Mix ingredients together
  Step: "Cook on a greased griddle \1FAE7\"
  Step:Serve
```

But this implied indentation might be considered significant, which it isn't. To be easy to use, white-space should be as irrelevant as possible. This is an equivalent single line (broken up only to fit on the page here):

```
Recipe:Title:Pancakes Ingredient:Name:Flour Amount: @unit:cups 2
Ingredient:Name:"Maple Syrup"Amount:@unit:tablespoon @approximate:yes "3"
Step:Mix ingredients together Step:"Cook on a greased griddle
\1FAE7\"Step:Serve
```

This is the complete set of syntax rules:

- `element-label:`
  - labels the start of an element's content
  - multiple character-sequence values of different types (see below)
- `@attribute-label:`
  - labels the start of an attribute's content
  - must a single character-sequence value, and may be the empty string when needed
- `"double-quoted character-sequence values"`
  - specify a sequence of character values that contain one or more white-space characters (space, tab, line feed, and carriage return)
- `` `back-quoted character-sequence values` ``
  - specify a sequence of character values that may or may not contain mark-down symbols for limited mixed-content and white-space
  - cannot be used for attribute specifications
- `unquoted-character-sequence-values`
  - specify a sequence of character values that does not contain spaces
  - all white space between two adjacent text values is reduced to a single space character
  - white space before the first text value and after the last text value is ignored
  - every unquoted text value must be followed by at least one white-space character
  - cannot be used for attribute specifications
- `\`
  - backslash escaping in unquoted and all quoted values (not in labels)
  - introduces the special handling of the limited set of following characters:
    - only `\"`, `\:`, `\@`, `\/`, and `\\` permitted for individual "sensitive" characters
    - used in mixed-content to escape the vocabulary's particular choice of mixed-content toggle symbols
    - `\{"0"-"9" | "A"-"F" | "a"-"f"}+\` produces a single Unicode character of the value of the hex characters escaped e.g. `\A0\` for NBSP, `\1FAE7\` for bubbles
- `/element-label`

- some schemas mandate the signalling of the end of element content to address schema structural ambiguity (see below)
- mixed content mandates the signalling of the end of element content

Except for the mandatory white-space character following unquoted text values, extraneous white space is entirely optional between elements. It can be used for indentation, it can be ignored in order to use a single line, or any combination in between. Significant white-space can be protected using quotes.

See https://github.com/CraneSoftwrights/AUTHORING.md for the latest implemented set of syntax rules.

3. What is already available?

At first the theory of getting any text into any XML to allow subsequent conversion into the target XML seemed to offer many potential paths to implement a transformation. And there are many tools out there with which one can get angle brackets from text:

AsciiDoc[3] - simple text to HTML, PDF, EPUB3, man(ual) page, DocBook.
- https://docs.asciidoctor.org/asciidoc/latest/
table.studio[4] - simple text to table structures
- https://table.studio
Markdown - no native XML output path
- (many flavours)
Pandoc[5] - generic plumbing; can convert Markdown to certain XML
- https://pandoc.org/

As XSLT cannot readily handle text input in the manner needed, it makes sense to use an existing tool to deliver an XML representation of the text stream and deal with that using XSLT.

However, the inference of end tags for descendent elements has to be algorithmically coded in a very imperative manner. This challenge has been examined by me for decades in areas such as splitting tables arbitrarily down the "middle" by finding all of the open elements, closing them (in order), and reopening them (in order) after the break. Another example is in breaking XSL-FO sequences of formatting objects down the "middle" to span over two page geometries, one in portrait and one in landscape.

In both real-world examples it proved untenable to support such arbitrary splitting imperatively. Limitations imposed at the highest level of the hierarchy precluded the need to have to deal with descendants. Crane's Page Sequence Master Interleave (PSMI) solution for leveraging XSL-FO page geometries is one

[3] https://docs.asciidoctor.org/asciidoc/latest/
[4] https://table.studio
[5] https://pandoc.org/

example that ended up getting packaged for others to use, when no other solutions were available at the time.

Determining from an input text stream that the arrival of content necessarily terminates a nested set of descendants probably can be done imperatively using a lot of logic, but the secondary challenge, then, would be to make it general purpose.

The iXML - Invisible XML - technology is more current than the above available technologies, but more precisely fits the transformation scenario than the off-the-shelf solutions listed above. In the grammar supplied the processor can recognize the hierarchy's obligations when following content is processed. The grammar declaratively itemizes the structured output to synthesize from input streams.

This simple iXML example of a small set of grammar rules illustrates how a formatted date string can be interpreted as XML structure:

```
date = day, " ", month, " ", year.
-digit = ["0"-"9"].
year = digit, digit, digit, digit.
month: "January"; "February"; "March";
    "April"; "May"; "June";
    "July"; "August"; "September";
    "October"; "November"; "December".
day: digit; digit, digit.
```

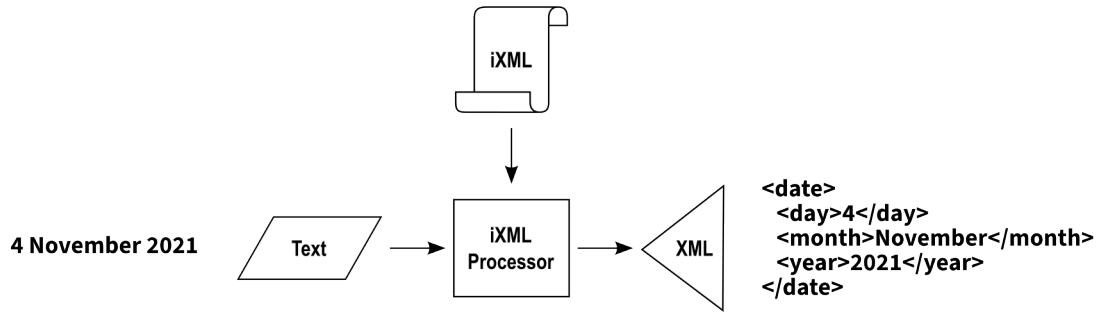
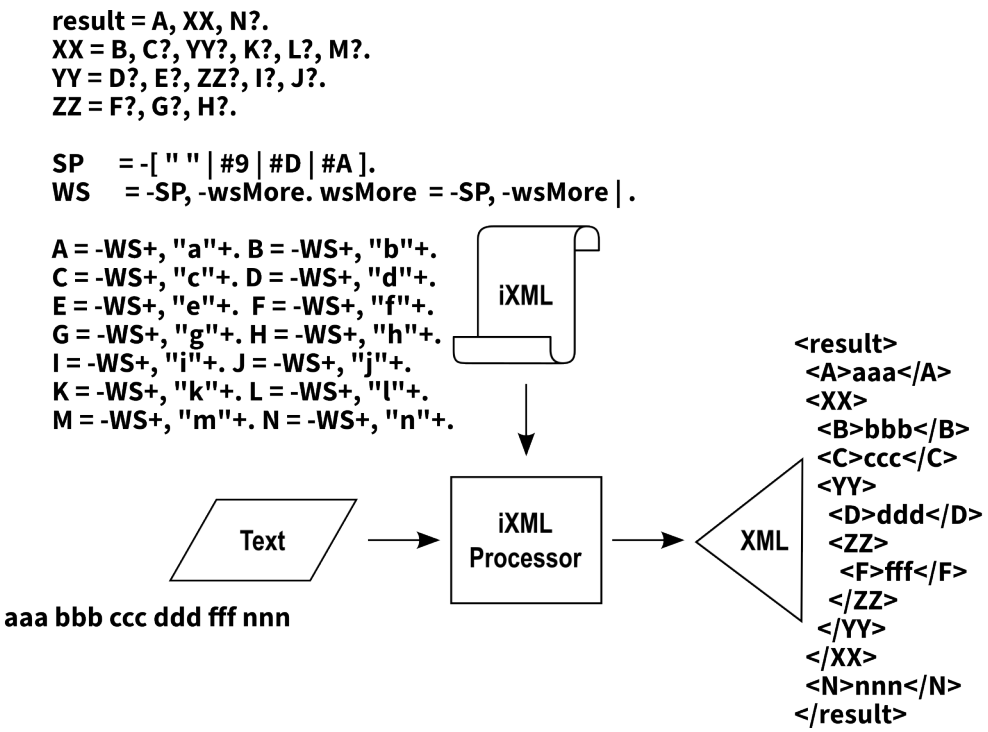


Figure 1. The iXML principle: a grammar transforms text into XML

This essence of supporting nested structure is key to solving the transformation to XML markup. Consider in the following when the iXML processor encounters the element `<N>` it automatically closes descendent elements `<ZZ>`, `<YY>`, and `<XX>` based declaratively on the rules for those elements, not on imperative logic:



The structure is expressed declaratively and interpreted by the processor. In an imperative solution, tracking all of the prior open descendants and determining to close each one would be very heavy lifting.

4. Implementation theory

What distinguishes the labeling of content in XML is whether or not the label represents a non-terminal element in the vocabulary (an item with element children) or a terminal element (an item with no element children). When using angle brackets in XML syntax, this non-terminal/terminal distinction is indicated using start and end tags.

The non-terminals are important to accommodate nesting. When non-terminals are known, that introduces tree hierarchy. With hierarchy, when the last of a descendant's content is written, the ancestral "opened" elements in between can be closed reliably.

The text-based Crane-txt2xml environment eschews (where permitted) end tags and relies on a combination of start labels and schema information to infer the hierarchy, thus taking the responsibility out of the editor's hands. But the environment is constrained to only those schemas that can be accommodated with the simple structures. This doesn't imply the structures are small, as this environment can accommodate in a single configured environment the 93 OASIS Universal Business Language (UBL) ISO/IEC 19854 schemas with over 5,400 elements in context.

So the struggle is not one of magnitude, but one of expressing XML structures in a non-XML manner for non-XML'ers to be effective in creating schema-valid documents while their domain-knowledge and expertise is focused on what matters to them.

One might consider using pattern replacement using regular languages (such as sed or regex), but even at a formal definition level these regular languages never can express the context-free structures needed for XML, thus everything is assumed to be a terminal. There is no state memory when applying regular languages.

Likewise, an iXML grammar with wild cards for element names (as first attempted in implementation) cannot distinguish between non-terminals and terminals, and so everything is assumed to be a terminal. What is required in this environment is an iXML grammar with knowledge of the target vocabulary document model schema.

5. The link to schemas

From the above recipe example XSD, it is possible to write the high-level grammar rules that recognize the text characters being converted:

```
  -__document_element = Recipe, -__WS*.
Title = __WS*, ( ( -"Title" ), -':' ), xs_string, ( __WS*, -'/',
( -"Title" ))?.
Name = __WS*, ( ( -"Name" ), -':' ), xs_string, ( __WS*, -'/',
( -"Name" ))?.
Step = __WS*, ( ( -"Step" ), -':' ), xs_string, ( __WS*, -'/',
( -"Step" ))?.
Amount = __WS*, ( ( -"Amount" ), -':' ), __content__AmountType, ( __WS*,
-'/', ( -"Amount" ))?.
Ingredient = __WS*, ( ( -"Ingredient" ), -':' ),
__content__IngredientType, ( __WS*, -'/', ( -"Ingredient" ))?.
Recipe = __WS*, ( ( -"Recipe" ), -':' ), __content__RecipeType, ( __WS*,
-'/', ( -"Recipe" ))?.
{continued with content definitions and low-level syntax}
```

There are six element types only, so the manual effort isn't insurmountable. But the pattern is readily programmatically synthesized.

The OASIS Universal Business Language (UBL) ISO/IEC 19854 vocabulary has over 4,000 elements in 5,400 contexts, defined requiring approximately 4,700 iXML rules. Clearly, the UBL grammar cannot be written by hand and must be synthesized.

This is the crux of this project: can iXML grammars be synthesized correctly from XML schemas to model a text conversion syntax that can be converted readily into schema-valid XML?

In fact, yes, the recipe grammar above was synthesized using the same stylesheets (described later) as now has been used to synthesize the UBL grammar.

But given the limitations for naming and structuring rules in iXML grammars, not all schema expressions can be converted readily to iXML. Given the iXML patterns are globally named without conflicts, this project focuses on Garden of Eden and constrained Venetian Blind schemas. Fortunately, rule marking allows element names and type names to be distinguished. As a reminder, these monikers were coined to distinguish a classification of schema expression based on whether elements and their types are globally or locally declared:

Table 1. Schema classifications based on declarations

Pattern classification	Element declarations	Type definitions
Russian Doll	Local (except for document element)	Local (anonymous)
Salami Slice	Global	Local (anonymous)
Venetian Blind	Local	Global
Garden of Eden	Global	Global

The most easily accommodated is any structure comprised solely of element content. Element content (with attributes) describes elements with either text content or other elements as content, but not a mixture of both. UBL models are only element content using the Garden of Eden declaration pattern.

Of course with effort one can convert any schema into a Garden of Eden schema by synthesizing and referencing global names in place of the local uses. The project includes the conversion stylesheet `Crane-salami2garden.xsl` that has been used to convert a Salami Slice schema into a Garden of Eden schema.

6. Environment configuration

The preparation of the authoring environment for a given vocabulary involves the synthesis of the iXML from the syntax of the document model. The expressivity of iXML pattern grammar semantics is similar to that of W3C Schema (XSD) and XML Document Type Definitions (DTD). Since a DTD is not expressed in XML, such a document model is converted to XSD using the trang utility. Recognizing that iXML cannot express certain Relax-NG (RNG) constraints such as interleave, the XSD approximation of RNG implemented in trang provides the best scenario for approximating what can be accommodated in RNG.

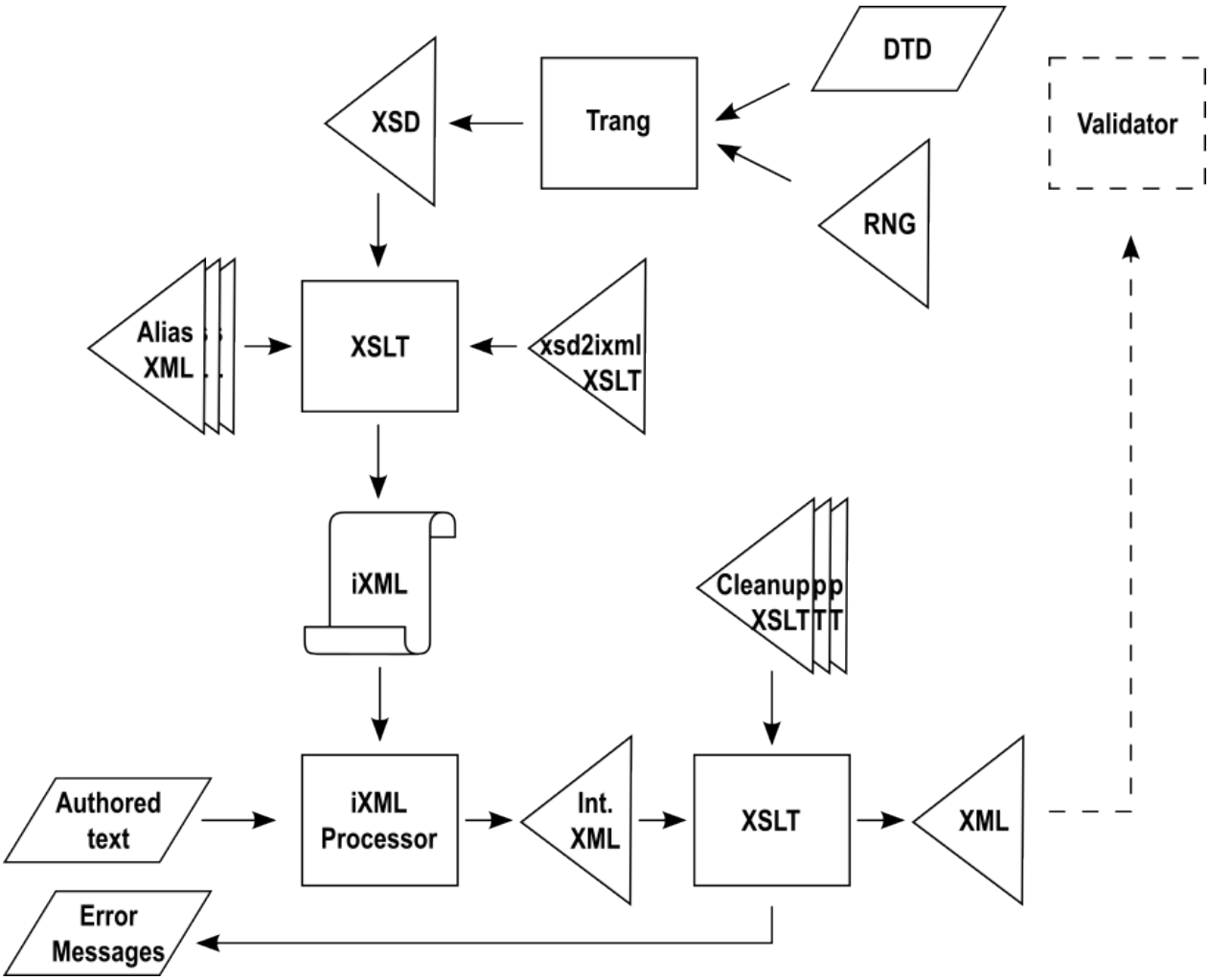


Figure 2. The configuration of the black box

As indicated earlier, not all document model structures can be accommodate in an iXML expression.

See https://github.com/CraneSoftwrights/IMPLEMENTING.md for the latest documentation on implementing a new vocabulary.

7. Simple user process

The author invokes a two-step process: converting their input into an intermediate XML as prescribed by the iXML grammar, then interpreting the intermediate XML into either error messages for the user, or output XML when there are no errors.

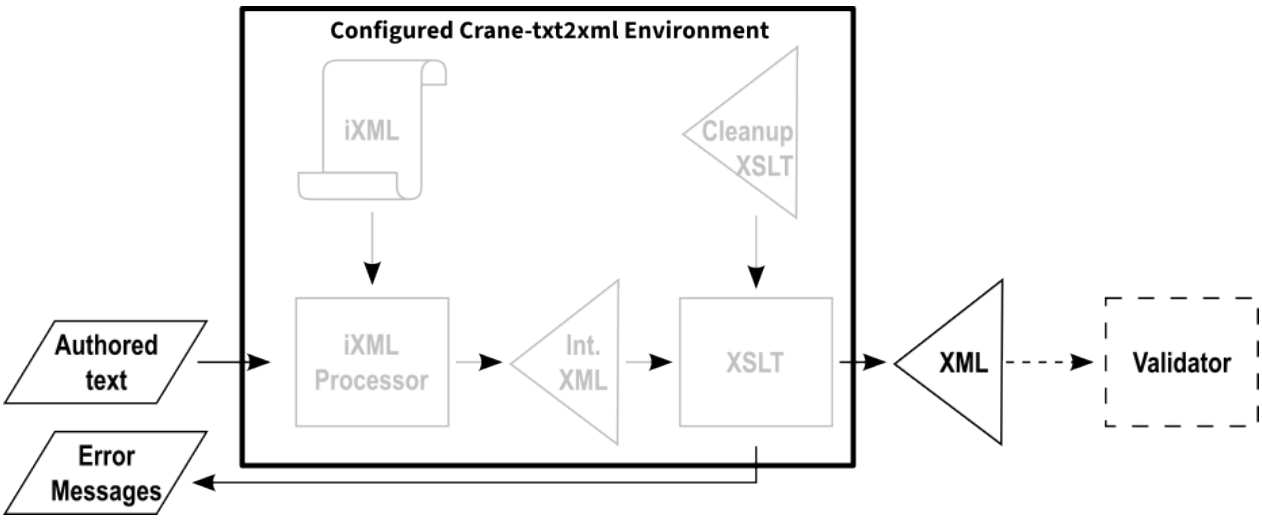


Figure 3. The author uses a black box

While the XML is well-formed, and structurally driven by the schema, it is possible that the character content is not appropriate. At this early stage of the project, XSD data types are not being validated at text conversion time. This is a possible future task if this environment is considered useful and evolves.

For the author, then, the follow-on step is to validate that syntax with the appropriate schema to evaluate the content constraints for the characters found in attribute and element content.

This isn't to minimize the importance of the users to know the vocabulary they are working with and the order of the content that gets wrapped in XML angle brackets. All the magic of iXML element closing cannot rearrange information into a new order. The user is obliged to use the correct labels for the elements in the correct order for the output to be produced.

The <PubNote> project includes command-line and drag-and-drop invocations of validation processes to support this important step.

8. Accessible labels for non-XML'ers

As described earlier, element and attribute labels are translated by this environment into element and attribute names. There is no programmatic reason that alternative expressions of labels cannot be used for any given element or attribute name. This has the power to make the data entry more accessible to non-XML'ers by configurations providing a catalogue of labels associated with XML element and attribute names.

And while XML names cannot contain spaces, the labels in this environment are permitted to include spaces. For example, each of these labels in a UBL-configured environment creates the `<WorkReportLineReference>` element:

- `WorkReportLineReference:`

- `Work Report Line Reference:`
- `WorkReport Line  Reference:`

Programmatically, this is accomplished for the thousands of UBL elements straightforwardly by breaking up camel-case words into words with any amount of intervening white-space.

While some environments offer labels, all environments always support the raw element and attribute names distilled from the schemas.

Consider this snippet of UBL input, focusing on the labels that signal the `<AllowanceCharge>` element:

```
PaymentTerms:
  Note: Penalty percentage 10% from due date
Allowance Charge:
  ChargeIndicator: true
  AllowanceChargeReason: Packing cost
  Amount: @currencyID: EUR 100
AllowanceCharge:
  ChargeIndicator: false
  AllowanceChargeReason: Promotion discount
  Amount: @currencyID: EUR 100
/Allowance Charge
TaxTotal:
  TaxAmount: @currencyID: EUR 292.20
  TaxSubtotal:
    TaxableAmount: @currencyID: EUR 1460.5
    TaxAmount: @currencyID: EUR 292.1
```

You can see two elements describing allowance charges. The first is labeled `Allowance Charge` and the second is labeled `AllowanceCharge`. The second one also has an end of content indication `/Allowance Charge` which does not have to match the start of content label. Note that end-of-content indicators are described later.

This label flexibility promotes imaginative opportunities for non-XML'ers responsible for creating what at times might seem to be cryptic XML names.

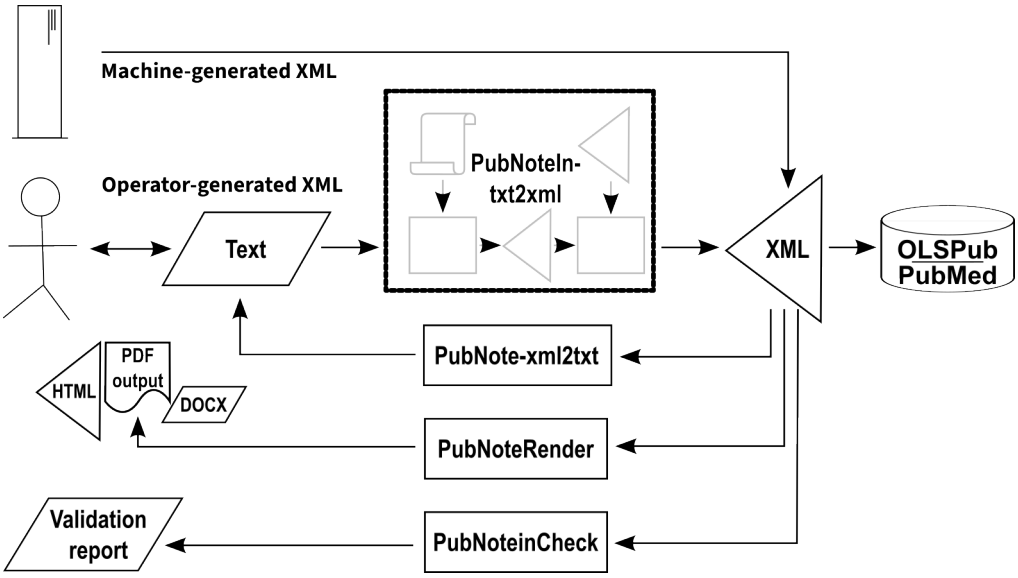
In the <PubNote> project, existing multilingual labels for presenting PubMed input content were repurposed programmatically to create the candidate label information needed for Crane-txt2xml. This permits the text output of <PubNote> to be direct text input to Crane-txt2xml, thus implementing a round-trip path from XML to text and back to XML.

For small publishers without existing support for XML, the <PubNote> environment provides:

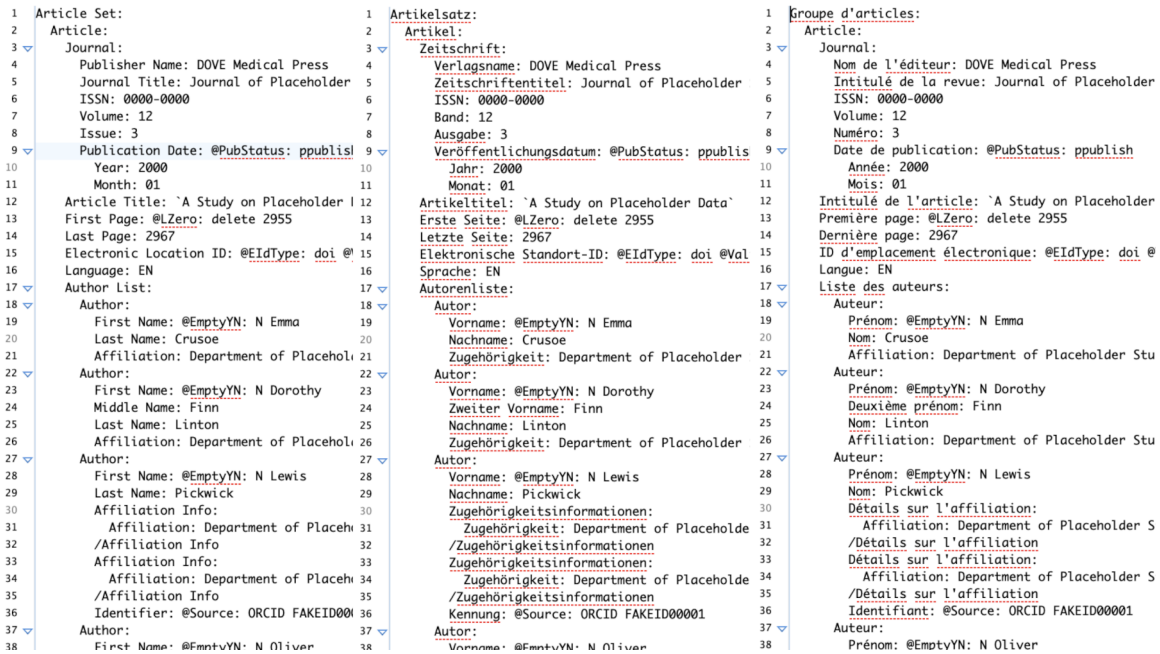
- a text-supported maintenance path for XML created either by machine or by hand,
- a text-origin creation path for XML,

- a rendering path to visually review XML content (PDF, HTML, and DOCX), and
- a validation path to programmatically review XML correctness.

The resulting XML conforming to the PubMedIn document model (available as a DTD) is ready to submit to OLSPub, all with open-source tools.



At the time of writing, <PubNote> supports the multilingual labeling of elements. Support for labeling all attributes in different languages should be added soon. Support for more languages is more than welcome. The following composite image illustrates three possible text files as input to PubNoteIn-text2xml-ZZ processes, all creating the same schema-valid XML output with English element names, expressed in English, German, and French:



```
Article Set:
  Article:
    Journal:
      Publisher Name: DOVE Medical Press
      Journal Title: Journal of Placeholder
      ISSN: 0000-0000
      Volume: 12
      Issue: 3
      Publication Date: @PubStatus: ppublis
        Year: 2000
        Month: 01
    Article Title: `A Study on Placeholder
    First Page: @LZero: delete 2955
    Last Page: 2967
    Electronic Location ID: @EIdType: doi @
    Language: EN
    Author List:
      Author:
        First Name: @EmptyYN: N Emma
        Last Name: Crusoe
        Affiliation: Department of Placehol
      Author:
        First Name: @EmptyYN: N Dorothy
        Middle Name: Finn
        Last Name: Linton
        Affiliation: Department of Placehol
      Author:
        First Name: @EmptyYN: N Lewis
        Last Name: Pickwick
        Affiliation Info:
          Affiliation: Department of Placeh
        /Affiliation Info
        Affiliation Info:
          Affiliation: Department of Placeh
        /Affiliation Info
        Identifier: @Source: ORCID FAKEID00
      Author:
        First Name: @EmptyYN: N Oliver
```

```
Artikelsatz:
  Artikel:
    Zeitschrift:
      Verlagsname: DOVE Medical Press
      Zeitschriftentitel: Journal of Placeholder
      ISSN: 0000-0000
      Band: 12
      Ausgabe: 3
      Veröffentlichungsdatum: @PubStatus: ppublis
        Jahr: 2000
        Monat: 01
    Artikeltitel: `A Study on Placeholder Data`
    Erste Seite: @LZero: delete 2955
    Letzte Seite: 2967
    Elektronische Standort-ID: @EIdType: doi @Val
    Sprache: EN
    Autorenliste:
      Autor:
        Vorname: @EmptyYN: N Emma
        Nachname: Crusoe
        Zugehörigkeit: Department of Placeholder
      Autor:
        Vorname: @EmptyYN: N Dorothy
        Zweiter Vorname: Finn
        Nachname: Linton
        Zugehörigkeit: Department of Placeholder
      Autor:
        Vorname: @EmptyYN: N Lewis
        Nachname: Pickwick
        Zugehörigkeitsinformationen:
          Zugehörigkeit: Department of Placeholde
        /Zugehörigkeitsinformationen
        Zugehörigkeitsinformationen:
          Zugehörigkeit: Department of Placeholde
        /Zugehörigkeitsinformationen
        Kennung: @Source: ORCID FAKEID00001
      Autor:
        Vorname: @EmptyYN: N Oliver
```

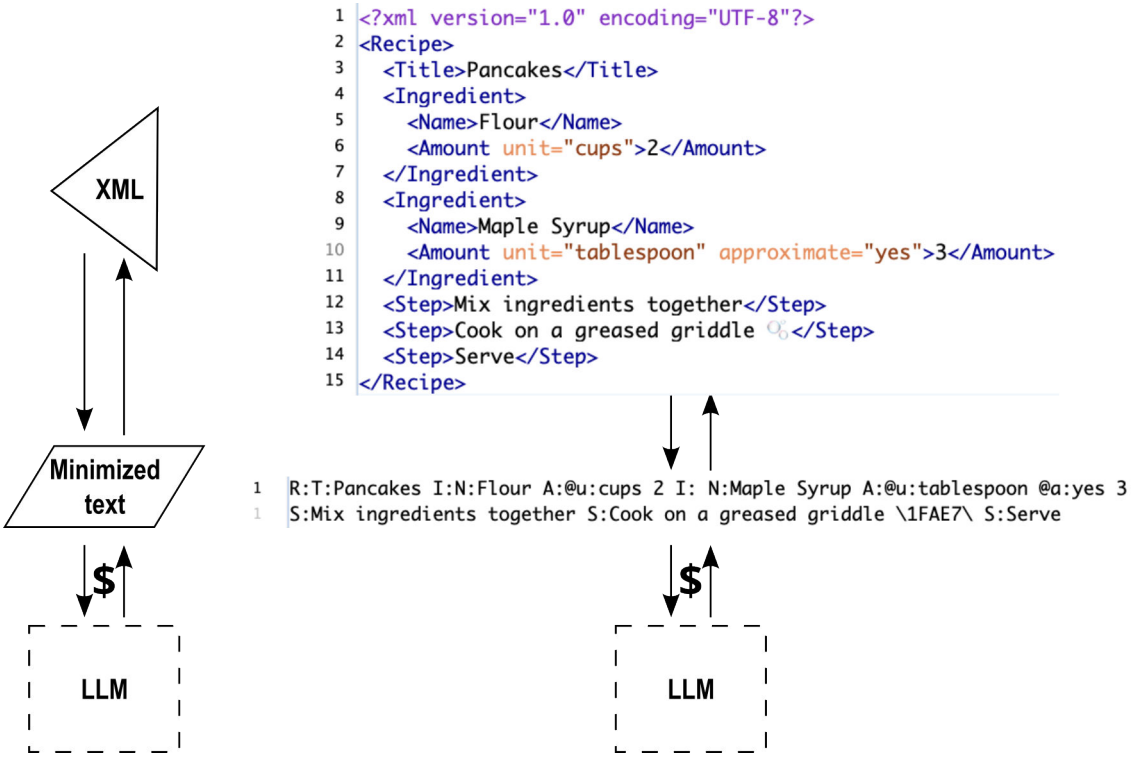
```
Groupe d'articles:
  Article:
    Journal:
      Nom de l'éditeur: DOVE Medical Press
      Intitulé de la revue: Journal of Placeholder
      ISSN: 0000-0000
      Volume: 12
      Numéro: 3
      Date de publication: @PubStatus: ppublish
        Année: 2000
        Mois: 01
    Intitulé de l'article: `A Study on Placeholder
    Première page: @LZero: delete 2955
    Dernière page: 2967
    ID d'emplacement électronique: @EIdType: doi @
    Langue: EN
    Liste des auteurs:
      Auteur:
        Prénom: @EmptyYN: N Emma
        Nom: Crusoe
        Affiliation: Department of Placeholder Stu
      Auteur:
        Prénom: @EmptyYN: N Dorothy
        Deuxième prénom: Finn
        Nom: Linton
        Affiliation: Department of Placeholder Stu
      Auteur:
        Prénom: @EmptyYN: N Lewis
        Nom: Pickwick
        Détails sur l'affiliation:
          Affiliation: Department of Placeholder S
        /Détails sur l'affiliation
        Détails sur l'affiliation:
          Affiliation: Department of Placeholder S
        /Détails sur l'affiliation
        Identifiant: @Source: ORCID FAKEID00001
      Auteur:
        Prénom: @EmptyYN: N Oliver
```

Figure 4. English, German, and French labeling in place of element labels

9. Concise labels to reduce LLM token ingest

The labeling facility in this environment offers the opportunity to use short cryptic labels instead of verbose natural language labels. This could provide an economical benefit in the world of working with Large Language Models.

Considering again the recipe example from earlier, using a single character as the label for each of the constructs reduces the 350-character (unindented) recipe XML instance as only 147 characters of simple text:




```xml
<?xml version="1.0" encoding="UTF-8"?>
<Recipe>
  <Title>Pancakes</Title>
  <Ingredient>
    <Name>Flour</Name>
    <Amount unit="cups">2</Amount>
  </Ingredient>
  <Ingredient>
    <Name>Maple Syrup</Name>
    <Amount unit="tablespoon" approximate="yes">3</Amount>
  </Ingredient>
  <Step>Mix ingredients together</Step>
  <Step>Cook on a greased griddle &#x1FAE7;</Step>
  <Step>Serve</Step>
</Recipe>
```

```
R:T:Pancakes I:N:Flour A:@u:cups 2 I: N:Maple Syrup A:@u:tablespoon @a:yes 3
S:Mix ingredients together S:Cook on a greased griddle \1FAE7\ S:Serve
```

This minimized representation could take far fewer tokens to convey to the LLM than raw XML syntax, but the information can be unambiguously parsed.

Of course that isn't the entire story, as an LLM already knows all of the iron-clad vagaries of XML syntax and would have to be taught the Crane-txt2xml authoring syntax using yet more tokens. Hallucinations might be prevalent with the novel syntax.

If this teaching has to happen with every prompt in the stateless system, the net benefit of the abbreviated labels is lessened. Though if the LLM cached the rules and exercised them, then that improves the equation for when this approach becomes practical and money-saving. Also, while this recipe is a single small instance, processing thousands of such instances would make the savings more substantial.

Passing the LLM output through one's own use of the Crane-txt2xml tool chain of grammar rules and XML model validation of the converted XML promotes data integrity.

10. Structural ambiguity

This project began with the intention of not ever needing to support the user specification of the end of content. After all, as described above, the iXML processor simply would close off descendent elements when ancestors implied no more content for the descendants.

What wasn't anticipated is the surprising presence of structural ambiguity, where the element cardinalities themselves present an ambiguous opportunity for the iXML processor to close off descendent elements.

Consider this simple contrived vocabulary:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="A" type="AType"/>

  <xs:complexType name="DType"/>

  <xs:complexType name="CType"/>

  <xs:complexType name="BType">
    <xs:sequence>
      <xs:element ref="C" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="AType">
    <xs:sequence>
      <xs:element ref="B"/>
      <xs:element ref="C" minOccurs="0"/>
      <xs:element ref="D"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="D" type="DType"/>
  <xs:element name="C" type="CType"/>
  <xs:element name="B" type="BType"/>

</xs:schema>
```

The element models are very simple: A contains B, then optionally C, and then D. B optionally contains C. Both C and D are empty.

This text input to transform into a fully populated instance reads as follows, without ambiguity, and so without any need for end-of-content indicators:

```
A: B: C: C: D:
```

This text input turns out to be structurally ambiguous because the element C could be either a child of B or a child of A, both validly:

```
A: B: C: D:
```

The explicit signalling of ending B before C unambiguously indicates that C is a child of A.

```
A: B: /B C: D:
```

The explicit signalling of ending B after C unambiguously indicates that C is a child of B.

```
A: B: C: /B D:
```

Without the explicit end-of-content signal any guess made by the transformation could very well be wrong for the user, so the user is obliged to signal what is needed.

At the time of writing, and only through testing, it has been identified once in the UBL vocabulary and once in the PubMedIn vocabulary. I did not attempt to programmatically identify all such structural ambiguities. Rather, and for symmetry, to cover future unidentified examples of structural ambiguity, the transformation grammar accepts the end-of-content signal for every label.

Note that where it has been identified, the configuration obliges the user to use the end-of-content indicator. Should a user encounter the necessity of the end-of-content signal, it is there to use and it can be reported as a GitHub issue for consideration as a mandated signal.

And when support for mixed-content was implemented, it became obvious that end-of-content indicators would be mandatory for those elements sitting as siblings of text content.

11. Mixed-content support

The popularity of Markdown has informed this project's initial approach to mixed-content.

The same labeling of the start and end of content applies to mixed-content, though the end-of-content indicators are mandatory. What becomes challenging for the writer of text is to enter the labels without violating any of the grammar rules that distinguishes the labels from surrounding text. This manifests as ambiguous parse errors that are difficult at times to diagnose and repair.

To simplify mixed-content data entry, the syntax provides a second way of quoting text for those elements that accept mixed-content: using back-ticks “”.

Within back-tick-quoted start and end ticks, the syntax recognizes typical Markdown symbols for whatever elements the configuration has mapped these too.

In the PubMedIn vocabulary, these mappings are provided:

- * - <b> boldface
- / - <i> italic
- ^ - <sup> superscript
- ~ - <sub> subscript
- _ - <u> underline

- `+ - <AbstractText>` elements, with applicable attributes

The need to support attributes in mixed-content limits the syntax for attribute specifications to quoted values and compact expression:

```
+@Label:"BACKGROUND"COPD has significant psychosocial impact. Self-
management
support improves quality of life, but programs are *not _universally_
available*.
IT-based self- management interventions can provide home-based support,
but
have mixed results.
```

...produces:

```
<AbstractText Label="BACKGROUND">COPD has significant psychosocial
impact.
Self-management support improves quality of life, but programs
are <b>not <u>universally</u> available</b>. IT-based self- management
interventions can provide home-based support, but have mixed
results.</AbstractText>
```

The flat nature of Markdown prohibits some of the expressiveness that might show up in XML found in the wild. Consider this example where the user creating XML puts bold XML inside bold XML (perhaps in error or perhaps under the mistaken impression the word "universally" might become extra bold):

```
<AbstractText Label="BACKGROUND">COPD has significant psychosocial
impact.
Self-management support improves quality of life, but programs
are <b>not <b>universally</b> available</b>. IT-based self- management
interventions can provide home-based support, but have mixed
results.</AbstractText>
```

This is not a problem in XML and it is not a problem when using the explicit labelling facility for mixed-content:

```
AbstractText: @Label: BACKGROUND "COPD has significant psychosocial
impact.
Self-management support improves quality of life, but programs are
 "b: "not "u: universally /u " available"/b . IT-based self- management
interventions can provide home-based support, but have mixed
results. /AbstractText
```

However, if this were converted mechanically into Markdown symbols, the resulting "`*`" characters lose their context and any expression of nesting/memory:

```
+@Label:"BACKGROUND"COPD has significant psychosocial impact. Self-
management
support improves quality of life, but programs are *not *universally*
available*.
```

```
IT-based self- management interventions can provide home-based support,
but
have mixed results.
```

And when this Markdown is converted to XML the asterisks get paired up differently than in the input:

```
<AbstractText Label="BACKGROUND">COPD has significant psychosocial
impact.
Self-management support improves quality of life, but programs
are <b>not </b>universally<b> available</b>. IT-based self- management
interventions can provide home-based support, but have mixed
results.</AbstractText>
```

… and the word "universally" ends up not being bold at all.

It becomes a user decision, weighing the complexity of the mixed-content with the expressivity of the Markdown limitations, when to use element labeling and when to use markdown for mixed content. Note that markdown cannot be used for element content.

12. Project challenges

12.1. Ambiguity

Many dozens of hours were invested in creating a syntax that minimized the opportunity for the lay user to trigger ambiguity problems beyond the previously-documented structural ambiguity.

The insurmountable challenges all were/are related to white-space. Eventually, simple rules were established to accommodate white-space between components of element content, though not without some constraining data entry rules. Ultimately, problems remain with white-space and mixed content.

The work continues to try and improve on this.

12.2. Conversion of a DTD to XSD

It is wonderful to have a powerful tool such as trang for the conversion of document models between different syntaxes.

However, the end result of the automated conversion wasn't purely in any of the four characteristic forms described earlier. It was necessary to manually tweak the resulting XSD so that the XSD to iXML conversion step would not be obstructed.

12.3. Documentation

Two areas are proving challenging in documentation for non-XML'ers as users:

- how to make the text-to-XML input syntax palatable, and
- how to translate the accurate but somewhat opaque iXML grammar violation errors into accessible error messages to make progress quickly in the editing of a file.

An attempt has been made to have as few special cases as possible so the user doesn't have to learn a lot to express what they need.

Of course there have to be rules for data entry, but the fewer rules the better. At all times when addressing grammar ambiguity and expressiveness, it was attempted to re-use syntax conventions to reduce what the user needs to learn.

Unfortunately, a grammar analysis cannot regroup and recover from a violation of the grammar rules when dealing with an input stream of characters. This obliges the user to fix one problem at a time, without being told of subsequent problems they could fix while they were in the editing process. Presenting grammar violations in an accessible fashion with precise pointers to user issues is essential in helping the user patiently tolerate the error recovery process.

End users are encouraged to file GitHub issues when they encounter challenges where the provided documentation falls short.

12.4. Configuration for new vocabularies

The Crane-txt2xml XSLT stylesheet fragments are designed for re-use, and the project package includes Crane-txt2ubl as an example adaptation.

As demonstrated in the <PubNote> project, a second adaptation for the PubMed input DTD using the PubNote-txt2xml stylesheet proves that the environment isn't constrained only to element-only content such as is used in UBL.

But examples are not guidance. More and better documentation is needed as guidance to implementers who are configuring their own environments using these components.

Configurations are encouraged to file GitHub issues when they encounter challenges where the provided documentation falls short.

12.5. Performance

The tool makers are doing a wonderful job supplying powerful and performant implementations of complex technologies. The project is being developed on an M4/128Gb computer providing fast execution and lots of elbow space.

The PubMed conversion is very usable. In less than 9 seconds conversions are made from text to XML. There are less than 400 iXML rules to parse all of the inputs.

The UBL conversion is not usable yet, but this isn't to disparage implementations. There are 4,700 rules in the UBL 2.4 grammar. Invoking the processor combines two steps: interpreting the rules and then executing the rules. I have

sent feature requests to tool manufacturers to support an intermediate interpretation of a set of rules in a "compiled" format so that execution focuses solely on applying the grammar to the text stream.

13. Conclusion

The power of iXML, in combination with the information available in a document model schema, provides enough information to configure a text-to-XML conversion environment. Such an environment guides the author through trial and error to produce properly-structured XML output. Further using the lexical constraint checking of the document model schema polishes the document into a conforming instance of the model.

The Crane-txt2xml environment is a platform on which implementers can create and customize the labeling of content to meet the needs of their non-XML-user base who may be discomfited having to deal with the vagaries of XML syntax.

Available free on GitHub, the community is invited to establish their own text-to-XML conversion environments based on Crane-txt2xml using UBL and PubMed as functional exemplars.

Proof that this project successfully socializes XML for non-XML'ers in these particular domains is in the hands of the users and their choice to end up using this work configured by experts for various environments.

Schematron 2025 – Technology Update

Erik Siegel
<erik@xatapult.nl>

Abstract

Schematron is a language for validating documents. It continues where schema languages like W3C XML Schema and Oasis RelaxNG stop. It allows you to check your documents against rules, usually expressed as XPath expressions, and define your own error messages. It is used a lot and integrated in several XML related products, for instance oXygen.

A new edition of the Schematron standard was published in September 2025, with many new features and enhancements. The presentation and this accompanying paper cover the most important changes. It also covers the tooling that can be used to apply, to actually use, Schematron 2025.

This paper and presentation is indebted to the excellent work of Andrew Sales, especially his comprehensive list of what's new in Schematron 2025: https://andrewsales.com/schematron4/index.html.

Keywords: XML, Schematron, Validation

1. Schematron in a nutshell

Here's an overview of Schematron's main high-level characteristics:

- Schematron is a formal schema language in which you can express rules for XML documents.
- There are two types of rules:
  - **Assertions:** when the condition for an assertion fails, an error message is issued.
  - **Reports:** when the condition for a report holds, a report message is issued.

  In practice, you will use assertions more often than reports.
- In Schematron you define all the error and report messages in your own words.
- Schematron is expressed in XML: a Schematron schema is an XML document.
- Schematron allows you to specify the underlying language for its expressions. In practice, XPath is the only language supported.
- Schematron can, by design, incorporate constructs from other programming languages. Currently support for XSLT and XQuery is available.

Schematron is an ISO standard and therefore not free. The official standard can be obtained from the ISO website at https://www.iso.org/standard/85625.html. Last time I checked it cost 204 Swiss franks. It's (as most standards) a formal and terse document that doesn't add a lot of value for normal day-to-day usage. My advice would be not to spend your money on it. There are enough free or much cheaper sources of information available (some of them listed at the end of this paper).

And what would a standard be without a mascot? And for Schematron, that's Schematroll:

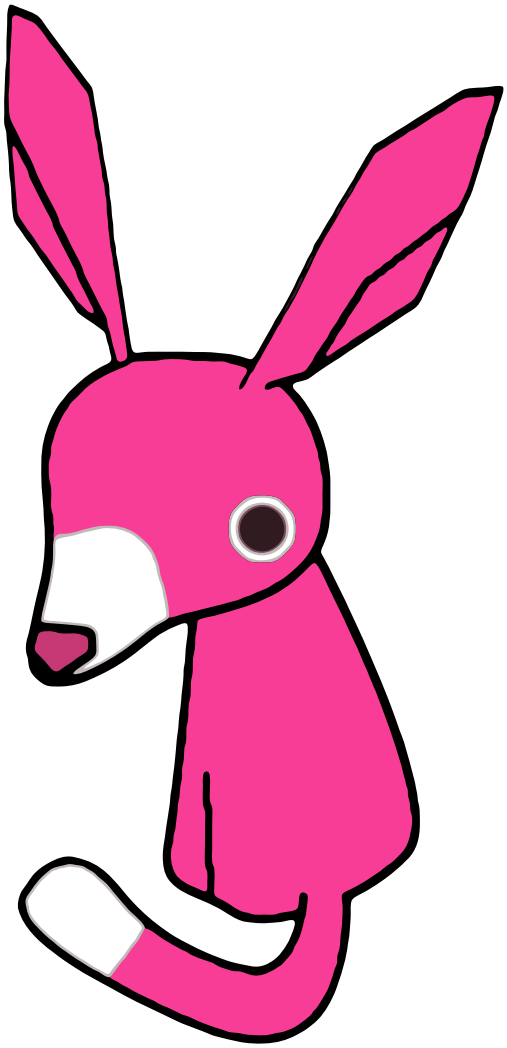


Figure 1. The notorious Schematroll

Schematroll was drawn by Cody Chang. It's a cross between two marsupials, the Bilby and the Bettong. You can use it for free on any Schematron-related product. It has its own GitHub repository: https://github.com/Schematron/schematroll.

1.1. Why Schematron?

There are a number of reasons why Schematron is such a useful tool in the XML toolbox. Here are the most important ones:

- Schematron is a relatively simple but powerful validation language. Basic Schematron already has a wide field of application and is relatively easy to master.
- Schematron can go way beyond the validations of the "classic" validation languages like DTD, W3C XML Schema, and RELAX NG. It allows you to do extensive checks on XML structures and data that are not possible in other languages. Anything you can express as an XPath test can be used for validation purposes. More experienced users can take advantage of XSLT features such as keys and functions.
- In Schematron you define all the error or report messages yourself. For other validation languages you're at the mercy of the validation processor's implementer, and this often results in technically correct but, for users, obscure messages. In Schematron this is completely under your control. Messages can be enriched with computed text from or about the validated document by using XPath expressions.
- Since the messages are under your control, Schematron is often used to partially take over validations normally done by other validation languages. Messages can be tailored to the user's knowledge level or context. So instead of:

  ```
  The content of element 'section' is not complete. One of '{para}' is
  expected
  ```

  You could tell the user:

  ```
  A section in a report must have at least one paragraph of text
  ```

1.2. An illustrative example

This example is a simple Schematron schema that checks an inventory list in an XML document for errors.

Schematron allows you to specify the underlying language for its expressions. In practice, XPath is the only language supported. There is nothing overly complex here, but to understand the examples, you need a basic understanding of XPath and namespaces.

Let's validate the following document:

Example 1. An example inventory list XML document

```
<inventory-list depcode="IMP">
  <article code="IMP0001">
    <name>Bolts</name>
```

```
    <description>Bolts to secure things with</description>
  </article>
  <article code="IMP0002">
    <name>Nuts</name>
    <description>Nuts to screw onto the bolts</description>
  </article>
  <article code="EXP0234">
    <name>Bananas</name>
    <description>Delicious ripe bananas</description>
  </article>
</inventory-list>
```

The rule imposed on this type of document is that the `code` attribute on an `article` element must start with the department code. This department code can be found in the `depcode` attribute on the root element. So the first two articles are valid, but the third one isn't.

Of course, this example is small, and you could easily validate it by hand. But what if it contained thousands of articles instead of just three? Or if you had many documents instead of just one. Automated validation then becomes necessary, which is where Schematron comes to the rescue.

1.2.1. Checking the value of the code attribute

Let's start by checking the value of the `code` attribute on the `article` elements in Example 1. The root element of the inventory list has an attribute for its department code called `depcode`. We need to verify that the value of each `code` attribute begins with this department code. For Example 1, the first two articles are valid, but the third one isn't. Example 2 validates this.

Example 2. A Schematron schema for checking the code attributes of Example 1

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">

  <pattern>
    <rule context="article">
      <assert test="starts-with(@code, /inventory-list/@depcode)">
        The article code must start with the right prefix
      </assert>
    </rule>
  </pattern>

</schema>
```

In a nutshell:

- All `article` elements in the document are checked, one by one:

  ```
  <rule context="article">
  ```

- The assertion that their `code` attributes start with the correct prefix is verified:

  ```
  <assert test="starts-with(@code, /inventory-list/@depcode)">
  ```

- If so, nothing happens, but if this isn't the case this message is issued:

  ```
  The article code must start with the right prefix
  ```

When validating Example 1 against the Schematron schema in Example 2, the third `article` element will raise an error, as expected:

```
The article code must start with the right prefix
```

This immediately illustrates one of the major benefits of Schematron validations: the message is in the code, it is part of the schema, and *you* specify the text.

1.2.2. Improving the message

The message issued by Example 2 can be improved. For instance, it would be nice if it would tell us what article it is about. For this, Schematron allows you to insert values from your document into your messages, using the `value-of` element (see Example 3). It also shows that you can use variables in a Schematron schema.

Example 3. Improving the message and using variables

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">

  <let name="department-code" value="/inventory-list/@depcode"/>

  <pattern>
    <rule context="article">
      <assert test="starts-with(@code, $department-code)">
        The article code (<value-of select="@code"/>) must start with
the
        right prefix (<value-of select="$department-code"/>)
        for <value-of select="name"/>
      </assert>
    </rule>
  </pattern>

</schema>
```

The validation message is now much more informative:

```
The article code (EXP0234) must start with the right prefix (IMP) for
bananas
```

2. Core language changes

This section describes the most important changes in the Schematron language version 2025, based on the 2020 version.

2.1. Specifying the Schematron edition

The root element for Schematron schemas `schema` has a new attribute called `schematronEdition`. This specifies the version of Schematron this particular schema conforms to. For the 2025 version, its value should be `2025`:

Example 4. Example of using the `schematronEdition` attribute

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025" …>
  …
</schema>
```

The attribute is optional for compatibility reasons. This makes sure that older Schematron schemas are still valid according to the 2025 specification. However, an edition 2025 Schematron schema should specify this.

2.2. Grouping of rules so all are evaluated

Schematron's assertions and reports are grouped within `rule` elements. For the Schematron 2020 edition, the only way to group `rule` elements was a `pattern` element.

A `pattern` element acts like an if-the-else statement: for every node the document being validated, only the *first* rule (in schema document order) for which the `context` attribute matches fires. For instance:

Example 5. Example of using the `pattern` element

```
<pattern>
  <rule context="book">
    … (asserts and reports for book elements)
  </rule>
  <rule context="*">
    … (asserts and reports for other elements)
  </rule>
</pattern>
```

For `book` elements the assertions and reports in the first rule are evaluated, for all other elements the assertions and reports in the second rule.

The Schematron 2025 version adds the `group` element for grouping `rule` elements. For a `group` element, the if-then-else behaviour of evaluating rules does not apply: it simply evaluates *all* rules contained. For instance:

Example 6. Example of using the `group` element

```
<group>
  <rule context="book">
    … (asserts and reports for book elements)
  </rule>
  <rule context="*">
    … (asserts and reports for other elements)
  </rule>
</group>
```

For `book` elements the assertions and reports in the first *and* second rule are evaluated, for all other elements only the assertions and reports in the second rule.

2.3. Specifying the severity of assertions and reports

The Schematron 2025 version introduces an additional attribute `severity` for `assert` and `report` elements, to express their relative importance. For instance:

Example 7. Example of using the `severity` attribute

```
<assert test="starts-with(@code, 'X')" severity="error">Invalid code</
assert>
```

The standard does not mandate values for this attribute. However, it reserves, without further definition, three values: `fatal`, `error`, `warning`, and `info`. This means that you are advised, but not obliged, to use these values.

The value for the `severity` attribute can be dynamically evaluated: you can state it as a variable reference (see also the next section).

2.4. Dynamic evaluation of the `flag`, `role`, and `severity` attributes

The `assert` and `report` elements can contain `flag`, `role`, and/or `severity` attributes. The Schematron 2025 edition allows the value of these attributes to be a variable reference. The value of this variable is then used as the value of the attribute. For instance:

Example 8. Example of dynamic evaluation of attributes

```
<let name="standard-severity" value="'error'"/>
```

```
…
<pattern>
  <rule context="@amount">
    <assert test="starts-with(@code, 'X')"
      severity="$standard-severity">Invalid code</assert>
  </rule>
</pattern>
```

A construction like the example above allows you to use the same, centrally defined, severity value in multiple asserts and reports.

2.5. Typed variables

The Schematron 2025 version now allows you to specify the data type of a variable, using the `as` attribute on the `let` element. For instance:

Example 9. Example of specifying the datatype for a variable using the `as` attribute

```
<let name="standard-severity" as="xs:string" value="'error'"/>
```

An XPath data type often uses the namespace prefix `xs`, which must be bound to the `http://www.w3.org/2001/XMLSchema` namespace. It depends on your Schematron processor whether it pre-defines this namespace prefix or not. If it doesn't, you'll get an error message about a missing namespace declaration when using type names like `xs:string` or `xs:boolean`. To avoid this, explicitly add the namespace declaration for the `xs` namespace prefix to your schema:

```
<ns prefix="xs" uri="http://www.w3.org/2001/XMLSchema"/>
```

2.6. Grouping abstract rules

Abstract rules are a mechanism for defining and reusing rules within a pattern or group. The Schematron 2025 edition introduces the top-level `rules` element, that allows you to define global abstract rules in your schema: abstract rules that can be reused in multiple patterns or groups. For instance:

Example 10. Example of using the `rules` element

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025">

  <rules>
    <rule abstract="true" id="abstract-rule">
      …
```

```
    </rule>
    <rule abstract="true" id="abstract-rule-2">
      …
    </rule>
  </rules>

  <group>
    <rule context="book">
      <extends rule="abstract-rule"/>
    </rule>
    <rule context="*">
      <extends rule="abstract-rule-2"/>
    </rule>
  </group>

  <pattern>
    <rule context="magazine">
      <extends rule="abstract-rule"/>
    </rule>
  </pattern>

</schema>
```

2.7. Libraries

The Schematron 2025 edition introduces a new mechanism for reusing Schematron schema declarations: libraries. A library is a separate document with a `library` root element. It may contain the same kind of content as a schema. However, it is not directly executable.

The contents of a library can be imported into another schema (or another library) using the top-level `extends` element. This element has an `href` attribute that references the library to include.

For example, assume we have the following library, called `codelib.sch`, that checks code attributes:

Example 11. Example of a Schematron library

```
<library xmlns="http://purl.oclc.org/dsdl/schematron">
  <let name="code-start" as="xs:string" value="'X'"/>
  <pattern>
    <rule context="@code">
      <assert test="starts-with(., $code-start)">
        A code must start with an <value-of select="$code-start"/>
      </assert>
    </rule>
```

```
  </pattern>
</library>
```

If we want to use these declarations in a schema, we have to reference the library using the `extends` element:

Example 12. Example of using a Schematron library with the `extends` element

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">

  <extends href="…/codelib.sch"/>

  …

</schema>
```

The net result is that the declarations in the library, excluding the `library` root element, are inserted at the location of the `extends` element:

Example 13. Example of the result of using a Schematron library with the `extends` element

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">

  <let name="code-start" as="xs:string" value="'X'"/>
  <pattern>
    <rule context="@code">
      <assert test="starts-with(., $code-start)">
        A code must start with an <value-of select="$code-start"/>
      </assert>
    </rule>
  </pattern>

  …

</schema>
```

2.8. External schema parameters

The Schematron 2025 edition introduces external parameters: An external parameter is similar to a top-level variable, but its value can be changed when invoking the schema.

To add an external parameter, use the `param` element as a direct child of the root element. It's definition is very similar to defining a variable: it has a required `name` and optional `value` attribute, with the same definition and semantics as for the `let` element. For example:

Example 14. Example of using an external parameter

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025">

  <param name="code-start" value="'X'"/>

  <pattern>
    <rule context="@code">
      <assert test="starts-with(., $code-start)">
        A code must start with an {$code-start}
      </assert>
    </rule>
  </pattern>

</schema>
```

In the example above, the external parameter `code-start` gets the default value `X`. When invoking the schema you can change this value. How to specify values for external parameters is processor dependent.

Unfortunately, the `param` element has no `as` attribute to specify its data type.

2.9. Defining parameters for abstract patterns and groups

Abstract patterns and groups allow you to re-use patterns and groups in your schema that contain structures that look alike but differ in things like, for instance, element names or other details.

Instantiating an abstract pattern or group happens using a `pattern` or `group` element with an `is-a` attribute. Values for the various parameters in the abstract pattern or group are passed using `param` elements:

Example 15. Example of instantiating an abstract pattern

```
<pattern is-a="table-pattern">
  <param name="table" value="table"/>
  <param name="row" value="tr"/>
  <param name="entry" value="td"/>
</pattern>
```

Abstract patterns for the 2025 edition must define the parameters they expect. This is also done using `param` elements, as the first children of the `pattern` or `group` element:

Example 16. Example of defining an abstract pattern with parameters

```
<pattern abstract="true" id="table-pattern">

  <param name="table"/>
  <param name="row"/>
  <param name="entry"/>

  <rule context="$table">
    …
  </rule>
  <rule context="$table/$row">
    …
  </rule>

</pattern>
```

Any `param` element in Example 16 could also have had a `value` attribute. This would define the default value for the parameter.

2.10. Changes to phase handling

Phases, specified using `phase` elements, are a mechanism to selective enable/disable patterns and groups in a schema. The Schematron 2025 edition introduces two changes for this mechanism: dynamic phase selection and partial document validation.

2.10.1. Dynamic phase selection

Dynamic phase selection is a mechanism to select which phase becomes active, based on the contents of the document that is validated. Two things have been added to the standard for this:

- The mechanism for selecting the phase to use has been extended with the special value `#ANY`. Specifying this triggers dynamic phase selection. How to pass such a value to the Schematron processor is implementation-defined.
- An optional additional attribute called `when` was added to the `phase` element. Its value must be an expression that can be evaluated/cast to a Boolean value.

Here is an example:

Example 17. Example of defining phases with dynamic phase selection

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
    queryBinding="xslt3">

  <phase id="validation-for-final-documents" when="/*/@status eq
'final'">
    <!-- Activate all patterns and groups for a final document: -->
    <active pattern="…"/>
    …
  </phase>

  <phase id="lax-validation" when="/*/@status ne 'final'">
    <!-- Activate all patterns and groups for a non-final document: -->
    <active pattern="…"/>
    …
  </phase>

  …

</schema>
```

Now if dynamic phase selection is active (the `#ANY` value was used), the expressions in the `when` attributes are evaluated. The first phase, in document order, where this attribute evaluates to `true` will become the active phase.

If no `when` attribute evaluates to `true`, all phases become active. This is the same as what happens when passing the value `#ALL` instead of `#ANY`.

2.10.2. Partial document validation

Partial document validation is a mechanism to restrict validation to subsets of the document, based on the active phase.

For this, the Schematron 2025 edition adds an optional attribute `from` to the `phase` element. Its value must be an expression that selects parts of the document being validated. When this attribute is present on an active phase, only the parts selected will be validated. Here is an example:

Example 18. Example of defining phases with partial document validation

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">

  <phase id="book-validation-phase" from="/*/book">
    <!-- Activate all patterns and groups for books: -->
```

```
    <active pattern="…"/>
    …
  </phase>

  <phase id="magazine-validation-phase" from="/*/magazine">
    <!-- Activate all patterns and groups for a magazines: -->
    <active pattern="…"/>
    …
  </phase>

  …

</schema>
```

2.11. Additional `visit-each` attribute for `rules` elements

The Schematron 2025 edition adds an optional attribute `visit-each` to the `rule` element. Its value must be an XPath expression that selects items, based on what was selected by the rule's `context` attribute. Each resulting item is validated against the assertions and reports of the rule.

Here is an example that checks whether words in a description that start with upper-case `CD` are followed by an integer number:

Example 19. Example of using a `vist-each` attribute

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
schematronEdition="2025"
  queryBinding="xslt3">
  <ns prefix="xs" uri="http://www.w3.org/2001/XMLSchema"/>

  <group>
    <rule context="description"
      visit-each="tokenize(., '[\s,.]')[starts-with(., 'CD')]">
      <assert test="substring(., 3) castable as xs:integer">
        Invalid code: <value-of select="."/>
      </assert>
    </rule>
  </group>

</schema>
```

Using `visit-each` theoretically allows processors and IDEs to better pinpoint the source of a problem. Whether this actually happens is implementation defined.

2.12. Base URI fixup handling

Base URI fixup is something users normally don't need to be aware of and can take for granted. Its formal definition can be found in the XInclude 1.0 standard, section 4.5.5 Base URI Fixup: https://www.w3.org/TR/xinclude/#base.

Its most important effect for Schematron is correctly handling relative document references, even if these are in included documents. A relative document reference, for instance `extends href="include/extra.sch"/`, must always get dereferenced from the document they're *in*, even if this is in an included document. The previous Schematron version did not define this and processor's often got it wrong.

2.13. language fixup

Language fixup has to do with the treatment of `xml:lang` attributes, which specify the natural language used in, for instance, assert and report messages. Its formal definition can be found in the XInclude 1.0 standard, section 4.5.6 Language Fixup: https://www.w3.org/TR/xinclude/#language

To find the natural language for the text in some element, a search will be done for the nearest `xml:lang` attribute, up from the current element up to the root of the document (following the XPath `ancestor-or-self::` axis).

2.14. Query language binding changes

Schematron supports query language binding (QLB). Query language binding allows you to specify the underlying programming language for, among other things, expressions in `let` and `value-of` elements. Some bindings, most notably XSLT and XQuery type bindings, allow you to add code in an additional programming language, thereby greatly expanding the scope of what you can do.

Theoretically, query language binding gives you a choice of an underlying language. In practice however, this is not the case. As far as I know, all publicly available Schematron processors support XSLT/XQuery/XPath type bindings only. The Schematron specification mentions a few others, but these are generally not supported.

The rules for the various query language bindings in Schematron have changed quite extensively in the 2025 edition. This section lists the major changes.

The query language bindings for EXSLT (`exslt`) and STX (`stx`) were removed from the specification.

2.14.1. Default query language binding `xslt`

The default query language binding `xslt` has changed as follows:

- The XSLT elements `xsl:import` and `xsl:include` are now allowed (before the first `pattern` or `group` element in the Schematron schema). This means you can now use any XSLT code in your Schematron schema, as long as it's in an external document.
- You can use attribute-value-templates (AVTs, XPath expressions between curly braces: `{…}`) for dynamic evaluation of the `flag`, `role`, and `severity` attributes. For instance: `severity="{if ($bad) then 'error' else 'warning'}"`

2.14.2. Query language binding `xslt2`

The query language binding `xslt2` has changed as follows:

- All changes as listed for the default `xslt` query language binding (see Section 2.14.1).
- The XSLT element `xsl:import-schema` is now allowed (before the first `pattern` or `group` element in the Schematron schema).

2.14.3. Query language binding `xslt3`

The query language binding `xslt3` has changed as follows:

- All changes as listed for the `xslt2` query language binding (see Section 2.14.2).
- Text value templates (TVTs, XPath expressions between curly braces: `{…}`) may be used as an alternative for `value-of` elements in messages. Whether and how this is supported is implementation-defined.

2.14.4. Query language binding `xpath31`

The query language binding `xpath31` is new in Schematron edition 2025. It is the same as the existing (and unchanged) query language binding `xpath3`, except that XPath 3.1 instead of XPath 3 is used.

2.14.5. Query language bindings `xquery3` and `xquery31`

The query language bindings `xquery3` and `xquery31` are new in Schematron edition 2025. Summary:

- Usage of XPath 3 (binding `xquery3` or XParth 3.1 (binding `xquery31`) for all expressions.
- You can use anything allowed in an XQuery prolog in your Schematron schema.

2.15. SVRL 2025 changes

SVRL (the Schematron Validation Reporting Language) in itself has not changed (much), but its definition and description in the standard has changed significantly:

- Where in the 2020 edition SVRL and its production was rather under-specified, there is now a full-blown description (in annex D of the specification) of how it should be produced given a Schematron schema execution.
- Small differences between Schematron and SVRL, for instance data types of attributes, were fixed.
- A new attribute `schematronEdition` was introduced for the SVRL root element `svrl:schematron-output`. Its value must be a copy of the `schematronEdition` attribute of the schema.
- An `svrl:error` element was added for reporting a dynamic error during Schematron validation. For instance an include file that wasn't found, a query language syntax error, etc.

3. Applying Schematron 2025

3.1. IDE-based Schematron validation

Suppose you have a Schematron schema (either written yourself or acquired) and some XML document(s) you want to validate. The easiest way to do this is by using an IDE (Integrated Development Environment) that supports Schematron validation:

- oXygen (https://www.oxygenxml.com) is a well-known and commonly used IDE in the XML world. Schematron edition 2020 is very well supported, including the Schematron QuickFix (SQF) extension. Unfortunately, at this time (June 2026) support for the Schematron edition 2025 was not yet available. oXygen has committed to supporting Schematron Edition 2025 starting version 29, scheduled for somewhere second half of 2026.
- XMLBlueprint (https://www.xmlblueprint.com/) is a lesser known and used XML IDE. However, it already supports Schematron 2025.

Both products are not free software, but both offer a trial license/period for testing things out.

3.2. The SchXslt2 Schematron processor

At the XML Prague conference in 2019, David Maus introduced a different way of doing Schematron validation in his talk, "Ex-post rule match selection: A novel

approach to XSLT-based Schematron validation.". This was the beginning of the rise of the SchXslt Schematron processor.

In 2020, SchXslt became the official Schematron processor and development/maintenance of the original "skeleton" processor stopped. SchXslt supports query language bindings `xslt` and `xslt2`. The SchXslt repository was archived in November 2025 and maintenance has stopped.

SchXslt was further developed and became SchXslt2. SchXslt2 supports the query language binding `xslt3` only and implements most of the new Schematron edition 2025 features. It has a number of manifestations:

- The core of SchXslt2 is a so-called "transpiler" (https://codeberg.org/SchXslt/schxslt2). This transpiler compiles a Schematron schema into an XSLT stylesheet. You can then use this stylesheet to validate your documents.
- You can use it directly from the command-line (https://codeberg.org/dmaus/mausotron-cli).
- There is an implementation of SchXslt2 as a step in XProc 3 (https://codeberg.org/dmaus/mausotron-xproc).

3.3. The XQS Schematron processor

XQS stands for "XQuery for Schematron" and is, as its name says, an XQuery implementation of Schematron. It was developed by Andrew Sales, the editor of the Schematron edition 2025 standard (and its previous version). It is a highly conformant implementation of Schematron edition 2025, supporting the query language bindings `xquery`, `xquery3`, and `xquery31`. You can find XQS on GitHub: https://github.com/AndrewSales/XQS.

XQS runs on the BaseX database engine (https://basex.org/, version 10 or higher). It operates using the same principle as SchXslt2: the Schematron schema is converted into, in the XQS case, an XQuery script. This generated script can then be used to perform the actual validation, using any conformant XQuery processor.

The XQS readme contains instructions on how to operate it. Don't let the requirement to use BaseX discourage you: once BaseX is properly installed (which is easy), using XQS is remarkably simple. No further knowledge of BaseX is required.

Be aware the XQS, because of its XQuery nature, does some things a little different than the usual XSLT based processors. Please read the XQS readme (also its Troubleshooting section), carefully.

4. For more information/references:

- Schematron in general:

  - The ISO standards page for edition 2025: https://www.iso.org/standard/85625.html

  - The Schematron website, maintained by the original designer of the language, Rick Jelliffe: https://schematron.com/

  - As it says itself: "A curated list of awesome Schematron tools and applications." Contains links to (all versions of) the specification, interesting articles, and more: https://github.com/Schematron/awesome-schematron

  - Andrew Sales is the editor of the previous (2020) and the current (2025) edition of the Schematron standard. His personal website contains various resources about Schematron, among which links to presentations and an extensive list with the 2025 updates (of which the author of this presentation and paper has gratefully made use): https://andrewsales.com/

  - My book about this new version of Schematron: Schematron 2025 - A language for validating XML. Available at https://xmlpress.net/publications/schematron-2025/

- Schematron supporting IDEs:

  - oXygen: https://www.oxygenxml.com/

  - XMLBlueprint: https://www.xmlblueprint.com/

- Schematron processors (supporting the 2025 edition):

  - SchXslt2 transpiler: https://codeberg.org/SchXslt/schxslt2

  - SchXslt2 from the command line (the "Mausotron"): https://codeberg.org/dmaus/mausotron-cli

  - XQS: https://github.com/AndrewSales/XQS

Implementing Maps for XPath 4.0

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

The main purpose of this paper is to describe how the Saxon product implements maps as currently proposed for the 4.0 versions of XSLT, XPath, and XQuery. The specifications introduce new features that create a number of challenges. In describing the implementation we also have to introduce these new features.

(Saxon is an implementation of the XPath, XQuery, XSLT, and XSD specifications. Saxon 12, released in 2023, is the version in general use: it implements a few of the early features in the 4.0 drafts. Saxon 13 is being released in 2026, with much more extensive coverage of 4.0 features. SaxonJ is the version for the Java platform; SaxonCS is the version for .NET.)

1. Introduction

Maps were introduced as a new data type in XSLT 3.0 and XPath 3.1. The original motivation came from the XSLT Streaming project: if you're going to process a large document in streaming mode, then you can't look back at parts of the document that you've already skipped over. This means when you encounter a piece of data that you're going to need later, you need to remember it somewhere, and that needs a more versatile data structure than the atomic values and nodes of XSLT 2.0. Maps were first proposed for XSLT 3.0 to address this requirement.

Maps are often used in conjunction with another XSLT 3.0 feature, accumulators: accumulators provide a declarative way of associating data with particular nodes in a document, defined as a function of the value of the accumulator on previous nodes, and data found at the current node. They correspond closely with the fold operation familiar from functional programming languages, except that they are based on iteration over the nodes of a tree structure rather than the items in a linear sequence.

Later on in the development of XSLT 3.0 and XPath 3.0/3.1, interest grew in allowing XSLT and XPath to process JSON, and it was realised that maps also had a big role to play there, because the data model for XPath maps included the model for JSON objects as a subset. Features provided by XPath 3.1 maps that go beyond JSON objects include:

- The key of an entry can be any atomic item, not only a string.[1]

The associared value can be any XPath value, including for example a node, a function, or a sequence of atomic items.

Maps have proved a popular addition to the language, and the draft 4.0 specifications [2] enhance that capability in a number of ways, based on user experience. Some of those capabilities are fairly superficial, in that they provide new functionality on top of the existing data model. But others have a profound effect on the design on the underlying data structures.

(I'll use the term XPath 4.0 for convenience to refer to the full set of 4.0 specifications, including XSLT and XQuery.)

The most significant new capabilities are:

- Maps in 4.0 are ordered.

  The main reason for making maps ordered is to make the serialized output of the map human-readable. With small maps (as with attributes on an XML element) a human reader can cope with variations in the order of entries, but when maps contain large nested maps, it becomes impossible to find your way around.

  The ordering applied to XPath 4.0 maps is essentially "first in, first out": new entries are added at the end. This should not be confused with sorted maps, where entries are maintained sorted in key order. In some cases (for example a map holding events indexed by timestamp), an application can use the map order to mirror the sort order of keys.

  Making maps ordered reflects their treatment in other modern programming languages. Javascript has guaranteed the order of entries in objects since ES2015 (though its rules are not the same as those in XPath 4.0), and Python has guaranteed the order of entries in dictionaries since version 3.7.

  The fact that maps are ordered clearly impacts the way they are implemented internally; a straightforward implementation using a hash table is no longer viable, because hashing does not preserve order.

- Maps and arrays in 4.0 can be treated as a tree of JNodes, and this tree can be navigated using path expressions and axes in much the same way as XML-based trees (whose nodes are now referred to as XNodes).

  The ability to navigate the tree using the full set of XPath axes is only possible because the nodes are ordered, but an implementation that preserves ordering is actually not enough. For example, in Java if you wanted a map that reflected order of insertion you would probably turn first to the class `java.util.LinkedHashMap`. But while a `LinkedHashMap` allows you to iterate over all the entries in the map in order, it does not allow you to find a specific

[1] XPath 4.0 uses the term *atomic item* where previous versions used *atomic value*.
[2] See https://qt4cg.org/

entry by key, and then navigate to the following or preceding entries, which is required to support the `following-sibling` and `preceding-sibling` axes.

Before describing how we address these requirements in Saxon, it will be useful to describe how we implement XPath 3.1 maps (and why).

2. Implementing maps in XPath 3.1

XPath is a functional language, and its data structures must therefore be immutable. Adding an entry to a map does not modify the original, it creates a new map and leaves the old one unchanged. For this to happen efficiently, it needs to use a data structure where entries can be added or removed incrementally without making a copy of the entire map. Data structures that enable this are variously referred to as *immutable* or *persistent* data structures, or sometimes *functional* data structures. I don't find any of these terms very satisfactory, because they all have different meanings in different contexts: let's instead call them *progressive* data structures. By a *progressive* data structure, I mean one that allows modification, but that returns the modified version as a new value while retaining the old value intact; and with the implication that such modifications can be achieved without the cost of making a full copy.

Saxon's primary implementation for XPath 3.1 maps, from Saxon 9.6 until Saxon 12, was the class `net.sf.saxon.ma.map.HashTrieMap`. On Java it was implemented using a third-party open source structure called `ImmutableMap` developed by Michael Froh; on .NET it was underpinned by Microsoft's `System.Collections.Immutable.ImmutableDictionary`. These both provide very similar functionality (though I've no idea how similar they are internally), so we needn't dwell on the differences. The basic idea of these structures is that the data is held in an internal tree, and when an entry is added or removed, a new tree is created that shares all the parts of the old tree that haven't changed (which usually means most of it). The cost of adding or changing a single entry is therefore independent of the size of the tree.

The underlying structure (on Java, Froh's `ImmutableMap`) isn't actually a map from XPath atomic items to XPath sequences, as one might expect. That's because the XPath 3.1 model requires us to treat two keys as being equal while also being able to distinguish them. For example, two QNames can be equal despite having different prefixes, and two `dateTime` values can be equal despite having different timezones, so we need to ignore the prefix or timezone when comparing keys, but to retain it when enumerating keys using the `map:keys()` function. It's also worth noting that neither the Froh nor the Microsoft implementations of the structure allow externally supplied `equals()` or `hashcode()` callbacks. In addition, the rules for comparing keys in an XPath map are not the same as the rules for the `eq` operator: comparison of map keys needs to be transitive, error-free, and context-free, and the `eq` operator has none of these properties. For

all these reasons, the key held in the underpinning `ImmutableMap` is not the XPath atomic item itself, but an `AtomicMatchKey` derived from it, chosen so that the `equals` and `hashcode` methods behave the right way for XPath maps. [3]

So much for the key part of the underlying `ImmutableMap`. The corresponding value is an instance of `net.sf.saxon.ma.map.KeyValuePair`, which as the name suggests is a pair of objects, an atomic item and a sequence. When we access a map to retrieve values by key, we're only interested in the sequence, but when we iterate over the entries in a map (for example, when evaluating `map:keys()`), we return the atomic item.

The `HashTrieMap` isn't the only implementation of XPath 3.1 maps in Saxon 12. There are other more specialised implementations, all conforming to the same Java interface so they can be used interchangeably. The most interesting variant is `DictionaryMap`, which handles cases where (a) all the keys are instances of `xs:string` (which means we don't need to store a type label with every entry), and (b) incremental modification using `map:put()` and `map:remove()` is unlikely. This is used, for example, for the maps that result from parsing JSON objects. If the application does decide to use `map:put()` or `map:remove()` on such a map, we take a hit by converting the `DictionaryMap` into a `HashTrieMap`.

There's one other little requirement that affects the design. What happens when you pass a map to a function that declares the required type of the argument as, say, `map(xs:integer, xs:QName)`? In a naive implementation, we would have to scan all the entries in the map checking that the keys are all integers and the associated values are all QNames. In some cases static typing might enable us to bypass this check, but those cases are probably rare. To handle this we do a little bit of optimization: as part of a `HashTrieMap`, we maintain fields for the "known key type" and "known value type". To check whether the map is an instance of a particular map type, we first check these values; if the known key type is a subtype of the required key type, and the known value type is a subtype of the required value type, then we need do no more. If not, we scan the whole map to establish a new known key type and known value type. The `map:put()` and `map:remove()` operations adjust the known type of the new map as necessary.

3. Implementing maps in XPath 4.0

We've discussed some of the new features of maps in 4.0: how does our implementation need to change?

[3] Wrapping an `AtomicValue` object in an `AtomicMatchKey` wrapper is potentially very space-expensive for a large map. However, for the common case of string keys, it enables an optimization: we can actually drop a layer of wrappers. Inside every `net.sf.saxon.value.StringValue` is a `net.sf.saxon.str.UnicodeString` object, and our `UnicodeString` interface implements `AtomicMatchKey` directly.

The first thing we did was to see whether we could find an existing implementation of immutable ordered maps that we could build on. Initially we thought we had found what we needed in the VAVR[4] project. This offers a `LinkedHashMap` that at first sight fits the requirement. Unfortunately we hit a performance bug[5] that made this a non-starter. One of the contributors to the VAVR project had fixed this with a redesigned class in a different way, and had made this available in a forked project, but there was little sign of the bug being fixed on the main branch. For a while we used the forked version, though we were a little uneasy about issues of support. But then the XPath spec introduced JNodes, and with it, the requirement to move from an entry to following and preceding siblings, and there seemed to be no immediate prospect of finding a third-party library that met that requirement, so we decided to think again.

Another factor here was that we needed solutions both for Java and C#. If VAVR didn't quite fit the bill in the Java world, we couldn't find anything remotely close for .NET.

It might be worth an aside here about the software engineering constraints involved in using open source component software. There have been some high-profile cases in the last year or two of big companies being hit by software vulnerabilities in open source libraries, and our customers are becoming increasingly anxious to seek evidence that we manage these risks, not just in our own software, but in our upstream supply chain. How do you go about reassuring yourself, let alone your customers, that a library you've downloaded from GitHub contains no malicious code? VAVR includes contributions from over 100 developers scattered around the world, and it contains about 50K lines of code. Where do you start? If the software had actually met requirements, we would have made the effort. But it didn't.

So, we started thinking about a do-it-yourself implementation.

Some observations that might influence the design (some of these have already been noted):

- Many maps are never modified after they are first built. A design that optimizes for this use case seems appropriate.

- More surprisingly, many maps are never used for retrieval by key. For example, when transforming JSON, one might parse a file containing thousands of JSON objects, and in most cases the vast majority of these will be either copied unchanged to the output, or discarded entirely. So there might be benefits to be obtained by lazy construction.

[4] https://github.com/vavr-io/vavr
[5] https://github.com/vavr-io/vavr/issues/2727#issuecomment-2567809521

- Most maps use string-valued keys. In many cases all the keys are of type `xs:string` (and not a subtype), so maintaining a type label for each entry is an unnecessary overhead.

- Most maps are small. Most of the options maps supplied to functions like `fn:xml-to-json()` contain only one or two entries. For a small map, it's quite acceptable to do a linear search to find an entry. In fact the 4.0 rules for options arguments say that it's an error to include an entry that's not defined in the spec, so the implementation has to scan the entire map anyway: indexing it adds no value.

- It's common for many maps to share the same set of keys. This is particularly true with the introduction of record types in XPath 4.0 (a record type defines a fixed set of keys, together with the required types of the corresponding values; records are maps and all map operations apply to records). But it also arises with other data, for example the `fn:parse-json()` function might well process a file containing a thousand maps each of which has exactly the same keys. Optimizing for this case would be useful.

- We may want to provide a range of possible implementations, and the best time to choose an implementation is when all the keys and values are known. This reinforces the point that it would be a good idea to provide a map builder that first simply gathers a list of key-value pairs, and only then decides what kind of map to construct.

So, whenever possible, we construct maps by supplying a list of key-value pairs to a mutable map builder; during this phase of processing all that the map builder does is to accumulate the list of key-value pairs.

For example, the first stage of building a map such as:

```
map{"LHR": "London",
    "LAX": "Los Angeles",
    "CBR": "Canberra",
    "YUL": "Montreal",
    "CIA": "Rome"}
```

is to construct a list of key-value pairs like this:

	Keys	Values
1	LHR	London
2	LAX	Los Angeles
3	CBR	Canberra
4	YUL	Montreal
5	CIA	Rome

Figure 1. Output of Map Builder

(Implementation details: the builder constructs two lists, an `ArrayList<AtomicValue>` for the keys and an `ArrayList<GroundedValue>` for the associated values. On completion, these are converted into two arrays, of type `AtomicValue[]` and `GroundedValue[]`.)

Before the map can be used, the only essential operation is to check the keys for duplicates. This can be done very cheaply by passing the keys through a Bloom filter. A Bloom filter typically maintains a bit array of say 65536 bits, and has a hash function that returns three independent numbers in the range 0..65535 for any key value[6]. Having applied this hash function to an input key, the Bloom filter checks whether all three bits in the bit array are set; if so, we have a potential duplicate. If not, we know that it cannot be a duplicate. If we're unlucky, and a possible duplicate is indicated, then we build an index of all the keys to make sure. [7]

At this point operations that scan over the entries in the map can be evaluated simply by iterating over the list of key-value pairs constructed by the builder. Many operations are now possible on the map without needing to build an index. For example:

- Functions such as `map:keys()`, `map:entries()`, or `map:for-each()` can be evaluated by scanning the list of key-value pairs.
- The value of `map:size()` is known.
- The map can be serialized using the JSON output method.
- It is possible to ascertain whether the map is an instance of a type such as `map(xs:string, node())`.

But suppose that the next thing that happens is a `map:get()` operation. We now need to build an index (except in the case where this was already built when

[6]The literature on Bloom filters gives formulae for optimizing the size of the bitmap and the number of bits that should be set. Our approach in comparison is somewhat rough and ready.

[7]Note that duplicate keys in a map under construction are not necessarily an error. There are various ways they can be handled, for example by taking the first duplicate, or the last, or by combining them into a sequence of values.

the Bloom filter detected possible duplicates). The index is an internal map from `AtomicMatchKey` keys (derived from the atomic items in the key list) to the integer offset of the key and value in the key-value-pair list constructed by the builder.

The structure now looks like this:

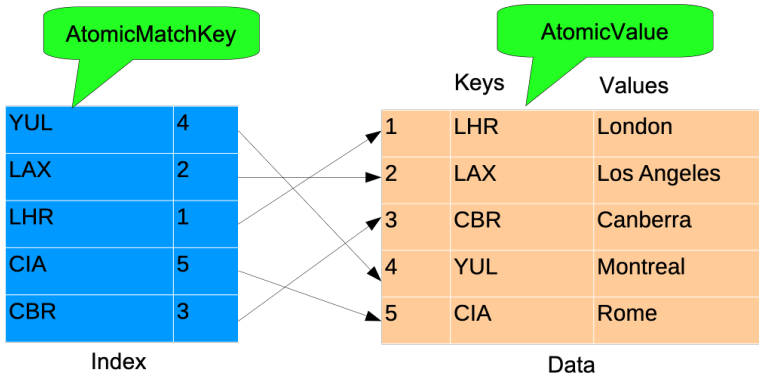


Figure 2. Solid Map with Index

Note that the index itself contains no ordering information, but the order of entries is maintained in the list of key-value pairs.

A `map:get` operation computes the `AtomicMatchKey` corresponding to the required atomic value, uses this to look up an integer offset, and then returns the value at that offset in the list of key value pairs.

(Implementation details: the index is a Java `HashMap<AtomicMatchKey, Integer>`. In this example, where the keys are strings, the `AtomicMatchKey` will be a `UnicodeString`; for strings consisting entirely of Latin-1 characters, the `UnicodeString` will be implemented as a `Twine8`, which wraps a `byte[]` array holding one character per byte.)

Now, if we're unlucky, someone will decide to do a `map:put` or `map:remove` operation. What happens now?

In our first iteration of the design, the first `map:put` or `map:remove` led to a bulk reorganization of the data structure. The list of key-value pairs was replaced by a progressive list (one that allows low-cost modifications without changing the original). The index from keys to offsets was replaced by a progressive map implementation (the same `HashTrieMap` that we used in Saxon 12). In order to avoid changing any integer offsets, `map:remove` did not actually remove the entry, but rather added its offset to a list of logically-deleted entries.

This design worked well for the vast majority of use cases, but we found it performed chronically badly in one or two applications. Specifically, we found an application that after constructing a map, added one single entry to it many times over. So we refined the design to avoid the bulk copy.

In the final design, a map in general consists of two parts: a *solid* part constructed by the builder which never changes once constructed, and a *fluid* part containing subsequent changes. The solid part is the structure we have already

seen: it consists of a list of key value pairs and an index to offsets in this list, neither of which will ever change, so it can be implemented using regular Java or C# structures (A `HashMap<AtomicMatchKey, Integer>`, an `AtomicValue[]` array, and a `GroundedValue[]` array).

The fluid part is essentially a list of changes that need to be applied to the solid part. It consists of the following components:

- A removals list: a set of integers, the offsets of entries that are no longer present because of `map:remove` operations.
- A replacements list: a map from integer offsets to values, identifying entries where the value associated with a key differs from the original value in the solid part.
- An additions list: this is similar to the solid part of the map: it contains a map from `AtomicMatchKeys` to integer offsets in a list of key-value pairs. The difference is that both the map and the list are progressive data structures, so incremental changes are possible.

The data structures used in the fluid part of the map are all progressive data structures, so they can be efficiently modified. The solid part represents the map as it was immediately after the initial build; the fluid part aggregates all subsequent changes.

Retrieval operations need to look in both parts. Update operations (`map:put` and `map:remove`) only affect the fluid part.

So if we add another airport to our example map, the structure will look like this:

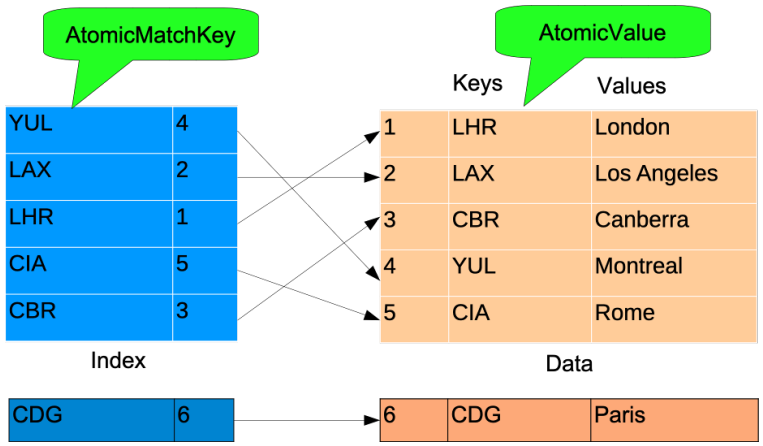


Figure 3. Modified Map

Retrieval operations now have to consider both parts of the map:

- `map:get()` looks first in the fluid part of the map; if it finds nothing, it looks in the solid part.

- Scanning operations such as `map:keys()` look first in the solid part of the map, skipping any keys that are also present in the fluid part; then the entries in the fluid part are appended.

There are a couple of subtleties here. The specification says that when `map:put` replaces the value for an existing key, the position of that entry in the map ordering does not change. So when we are scanning, then for each key-value pair that we encounter, we must skip it if it is present in the removals list, and we must return the replacement key and value if it is present in the replacements list. The specification leaves it implementation-defined whether a `map:put` operation replaces the key as well as the value (consider the case where `map:put` changes the value associated with a given QName, but supplies as a key value a QName with a prefix that differs from the original). The Saxon implementation retains the original key value.

In a pathological case, for example a map holding timed events where new events are continuously added and old events are continuously removed, we might get to the point where every single entry in the solid part has been removed or replaced, and in that case we can reorganize the map.

4. Performance Measurements

The main performance objective in this exercise was to ensure that introducing ordered maps did not cause a significant performance regression between Saxon 12 and 13. We have run a few simple performance measurements to convince ourselves of this; this doesn't claim to be a comprehensive analysis, and there are many external factors that might distort the result. But from the figures given below, the results give confidence.

First I have measured map construction using a MapBuilder versus incremental construction using successive calls of `map:put`. This was done using direct calls of the Java implementation classes, not by means of XPath expressions, in order to avoid extraneous factors such as XPath parsing cost intruding on the measurements.

For map sizes of 1000, 10,000, and 100,000 entries, with Saxon 13 timings in microseconds for bulk construction using a MapBuilder were 193, 478, and 9210μs respectively. Corresponding timings for incremental construction using a sequence of `map:put` operations were significantly slower: 240, 1949, and 29,368μs. (The test case built the map and then did a single `map:contains` operation to force indexing.)

In Saxon 12 the equivalent figures for bulk construction were 214, 2122, and 37,699μs, and for incremental construction 138, 1527, and 20,561μs.

So at first sight, bulk construction in Saxon 13 with the new design (which is likely to be the dominant mode of processing) is significantly better than in Saxon 12, while incremental construction is perhaps a bit slower, but not worryingly so.

In Saxon13, making one million `get` requests on a map with one thousand entries took 29,423µs. The corresponding figure with Saxon 12 was 29,245µs - effectively identical.

In Saxon13, evaluating `map:keys` on a map with one hundred thousand entrues took 9092µs. The corresponding figure with Saxon 12 was 40,978µs. Scanning a map sequentially can be expected to be considerably faster with the new data structure, since the keys are held in a single list, and have good memory proximity leading to good CPU caching.

The above figures are all for SaxonJ. There is no reason to expect a significant difference with SaxonCS.

5. Shaped Maps

When many maps share the same sequence of keys, but with different values, then it makes sense to hold the sequence of keys once, rather than repeating it in every map.

Saxon 13 implements a structure called a `ShapedMap` that works this way. The list of keys is called a `Shape`, and the `ShapedMap` consists essentially of a reference to the `Shape`, plus an array of slots holding values of the entries.

A `ShapedMap` does not have any kind of index. Most such maps are small, and a sequential search for the key can be expected to perform well. In addition, lookup expressions that reference the key statically (for example `$person?name`) are very common, and these expressions use a simple caching mechanism: if in one evaluation `$person` is a `ShapedMap`, and in the next evaluation `$person` is another `ShapedMap` with the same `Shape`, then it knows that the offset in the array of values will also be the same as last time.

Saxon 13 uses shaped maps only for records, corresponding to either a built-in record type defined in the language specification, or a user-declared record type. There is scope to extend their use to other use cases, notably to the maps that result from JSON parsing. This requires the parser to recognize that multiple maps are using the same structure.

In fact, the general design for 4.0 maps, with its use of a list of key-value pairs, can easily be extended to combine with the general idea of shaped maps. Since the general structure uses a HashMap from keys to integer offsets, and the keys are the same in multiple maps, this HashMap can be shared. This is a potential development beyond Saxon 13.

6. Typing and Coercion

XPath allows an expression of the form `$map instance of map(K, V)`, testing whether all the keys in a map are of type `K` and all the values are of type `V`. Such a test is also performed implicitly when a map is passed as an argument

to a function that expects a map of a particular type. This test is potentially very expensive, involving a full scan of the entries in the map.

We attempt to reduce this cost through caching. If no cached data is available, we evaluate the instance-of test by examining every entry in the map. We then record the fact that the map is known to be an instance of `map(K, V)`. This information is retained (or suitably amended) in any new map constructed using `map:put()` or `map:remove()`. If there is a test for some other unrelated map type `map(K2, V2)`, then we potentially re-evaluate the condition from scratch; this is hopefully rare. But it's theoretically possible, for example, that a map that wasn't an instance of `map(K, V)` becomes valid against that type after a sequence of `map:remove()` operations, and we need to allow for this.

When a map is passed to a function that requires an argument of type `map(K, V)` then the language spec requires the system to coerce the supplied map to the required type, which may potentially involve coercing all the keys (or all the values) individually. We're looking at how to do this operation lazily to reduce the cost, but we don't yet have a really efficient solution to this requirement. The requirement is particularly onerous when the map contains function items, because in this case function coercion is needed even if the supplied function is a subtype of the required function type.

A. JNodes

As mentioned, XPath 4.0 introduces the idea of being able to navigate a tree of maps and arrays using path expressions that are very like the traditional paths used to navigate XML-based trees of elements, attributes, and text nodes. The key to making this possible is the idea of a tree of JNodes, where the JNodes encapsulate the maps and arrays and their entries and members.

JNodes are described in [Kay 2025], and more fully, of course, in the draft 4.0 specifications. But a summary might be appropriate here, since it imposes requirements on the design of maps as presented in this paper.

An array can be wrapped in a JNode. It is then possible to navigate to the children of the JNode, which correspond to the members of the array. The JNode that wraps one of these children retains not only the value of the relevant array member, it also retains a property identifying the parent JNode (wrapping the original array), and the member index within the array. This makes it possible to navigate upwards from one of these child JNodes back to the parent, and also sideways to the siblings, representing the other members of the array. If the value of the array member is a map or array, then it is also possible to navigate downwards, to the grandchildren of the original array.

Similarly, a map can be wrapped in a JNode. Navigating to the children of this JNode delivers a set of JNodes that correspond to the entries in the map (retaining order). These child JNodes contain not only the value of the map entry,

but also the key, and a reference to the parent map. So again, navigation becomes possible upwards and sideways as well as downwards.

All the other axes (with the exception of the attribute and namespace axes) are defined as combinations and closures of the basic relations of a node to its children, parent, and siblings, so all these XPath axes become equally applicable to trees of maps and arrays.

This is achieved without giving maps and arrays a persistent identity. This means that two maps with the same entries can always be treated as being the same map, which is what makes it possible for a small modification to a large map to retain the parts of the map that have not changed: small modifications to a large tree take constant time.

B. The Journey: How did we get here?

The design presented in this paper is the current stage of a long journey, many of whose steps have been documented in previous conference papers. There have been many false turns along the way. Rather than a traditional list of references, I would like to give an account of this journey, highlighting which ideas worked, which were discarded, and which were heavily amended in the light of experience.

My first exploration of this area was in [Kay2007] where I explored the possibility of writing an XSLT optimizer in XSLT. It seemed to me that optimization is largely a matter of transforming expression trees, and since transforming trees is what XSLT is all about, the idea ought to be feasible. I quickly discovered that if the trees were implemented as XNode trees then it wasn't possible to achieve anything like good enough performance. The basic reason is that the design of XNode trees (in particular the fact that XNodes have node identity, and a reference to their parent) makes it very hard to find a way of making a small modification to a large tree in constant time.

I returned to this theme in [Kay2018a], where I explored the idea of a K-Tree, a tree where nodes were materialized on demand in the course of navigation [1], enabling the result tree of a transformation to share subtrees of the source tree if they had not been modified. The practicalities of implementing this proved too complex: considerations of node identity, document order, namespace context and the like meant that no performance benefits were achieved in practice. Although the design did make some operations substantially faster, the more basic and common operations of tree navigation all became a bit slower because of the extra complexity.

However, the experience was useful, in several ways.

[1] An idea I subsequently learned was well known in functional programming circles, and was referred to as a Zipper [Huet1997]

- Firstly, we achieved some significant applications that used XSLT to transform code.

  One was the XX compiler, a compiler for XSLT, generating SEF files for execution in the browser using SaxonJS: this is described in [Kay&Lumley2019]. Acceptable performance was achieved primarily by reducing the number of transformation phases in the compiled pipeline, and doing more work in each phase.

  Another was the transpiler that we use internally to convert the Java code of the Saxon product into equivalent C# code: the transpiler is written primarily in XSLT, and is described in [Kay2021]. This is essentially a four-pass transformation. The first pass (written in Java) converts the Java syntax to XML. The second pass constructs a digest of the source tree, building an index of all the packages, classes, and methods. The third pass analyzes this digest and decorates it with information needed to generate the C#, for example by determining which methods should be receive `override` or `virtual` modifiers. The final pass takes the entire XML representation of the syntax produced in phase 1, and serializes it as C#, using the information found in the digest.

  Although the transpiler was based on transformations of XML trees, and is used daily in production in that form, I also prototyped a redesign using maps and arrays. This case study contributed many of the ideas for the 4.0 specification of maps and arrays, ideas which I presented in [Kay2025a]. That paper preceded the introduction of JNodes, but reading the conclusions of the paper, the essential ideas for JNodes were all there, and led directly to the introduction of JNodes at [Kay2025b].

- The idea behind JNodes [Kay2025b] was strongly influenced by my work on the K-Tree design [Kay2018a], and the problems in implementing it, caused by node identity, namespace context, and parentage, and it reused the idea of building nodes with parent pointers dynamically in the course of tree navigation, enabling a modified tree to share subtrees with the original and thus allowing modification in constant time. Awareness of these problems explains why I successfully argued during the development of XSLT 3.0 and XPath 1.0 that maps and arrays should not have identity or parent pointers, and this design decision can be seen as paving the way to making JNode navigation possible.

References

[1] . Gerard Huet. *The Zipper*. Journal of Functional Programming. 7 (5): 549–554 doi:10.1017/s0956796897002864. S2CID 31179878.

[2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages, Montreal, 2007. Available at `http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html`.

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

[4] Michael Kay. *XML Tree Models for Efficient Copy Operations*. XML Prague 2018. Available at `https://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf` and at `https://www.saxonica.com/papers/xmlprague-2018mhk.pdf`.

[5] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Markup UK 2018. Available at `https://markupuk.org/2018/Markup-UK-2018-proceedings.pdf` and at `https://www.saxonica.com/papers/markupuk-2018mhk.pdf`.

[6] Michael Kay and John Lumley. *An XSLT compiler written in XSLT: can it perform?*. XML Prague 2019. Available at `https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf` and at `https://www.saxonica.com/papers/xmlprague-2019mhk.pdf`.

[7] Michael Kay. *<transpile from="Java" to="C#" via="XML" with="XSLT"/>*. Markup UK 2021. Available at `https://markupuk.org/2018/Markup-UK-2021-proceedings.pdf` and at `https://www.saxonica.com/papers/markupuk-2021mhk.pdf`.

[8] Michael Kay. *XSLT Extensions for JSON Processing*. Balisage 2022. Available at `https://www.balisage.net/Proceedings/vol27/html/Kay01/BalisageVol27-Kay01.html` .

[9] Michael Kay. *Processing JSON with Template Rules*. Markup UK 2025. Available at `https://markupuk.org/webhelp/index.html#ar04.html`.

[10] Michael Kay. *JNodes: a New Model for Navigating JSON Trees*. Balisage 2025. Available at `https://balisage.net/Proceedings/vol30/html/Kay01/BalisageVol30-Kay01.html` .

Integrating AI into XML Development Workflows

Octavian Nadolu
Syncro Soft / Oxygen XML Editor
<octavian_nadolu@oxygenxml.com>

Abstract

AI has moved quickly from a curiosity to a feature that many software teams now expect to find in their daily tools. XML teams are no exception, but their situation is different from that of many general programming environments. XML work is shaped by schemas, namespaces, reusable content, transformation pipelines, publishing constraints, and long-lived conventions. A generated answer that looks reasonable may still break a link, miss a namespace, or silently change publishing behavior.

This paper looks at AI support for XML development from the point of view of everyday editor-based work. It discusses where current systems help most: producing first drafts, explaining unfamiliar code, suggesting completions, reviewing existing artifacts, and coordinating repeatable project tasks. It also separates ordinary generative assistance from agentic workflows, where the assistant plans steps, calls tools, checks results, and revises its work.

The central argument is deliberately modest. AI should be allowed to help with interpretation, drafting, and planning, while XML-aware tools continue to handle validation, refactoring, querying, testing, and publishing checks. The paper uses examples from XSLT generation, context-aware completion, XML and XSLT review, cross-artifact element renaming, and DITA map restructuring to show where this division of responsibility works well. It closes with common failure modes and adoption practices for teams that want the benefit of AI without losing correctness, traceability, or human control.

1. Introduction: Why AI for XML?

XML is still part of the infrastructure of many publishing, data interchange, compliance, and enterprise integration systems. The people who maintain those systems work with schemas, XPath, XQuery, XSLT, Schematron, DITA, DocBook, and related technologies. These technologies are powerful, but they are also exacting. A small change in a namespace declaration, a match pattern, or a reused fragment can have consequences far away from the file being edited.

Recent large language models have made it possible to bring assistance directly into this work. An editor can now help draft a schema component, explain an XPath expression, sketch an XSLT template, propose a Schematron assertion, summarize a legacy stylesheet, or review a change before it is committed. For experienced XML developers, this can remove some routine effort. For newcomers, it can make an unfamiliar project less opaque.

The risk is that the same fluency that makes these systems useful can also make mistakes easy to miss. XML workflows are not tolerant of near misses. A plausible XPath may select from the wrong context. A generated stylesheet may work for the sample file and fail for the next one. A refactoring that updates the instance documents but not the CSS, tests, or documentation may leave the project inconsistent. AI is useful in XML work when it speeds up expert practice, not when it replaces the practices that keep XML projects safe.

This paper is therefore not a benchmark of model quality. It is a practice-oriented discussion of how AI can fit into XML engineering. It identifies useful tasks, distinguishes between generative and agentic forms of assistance, proposes a bounded integration model, and summarizes lessons from editor-centered scenarios where AI and XML-aware tooling are used together.

2. From Generative Assistance to Agentic XML Workflows

It is helpful to distinguish between **generative AI** and **agentic AI**. Generative AI produces an answer to a prompt: a code fragment, an explanation, a summary, a translation, or a suggested rule. In XML development, this may mean a first version of an XSLT template, a draft Schematron assertion, a short explanation of a schema fragment, or a summary of a DITA topic.

Agentic AI tries to complete a task rather than simply return text. It may search the project, inspect schemas and stylesheets, edit several files, validate the result, correct problems, and then report what changed. The important difference is not the use of a larger model, but the addition of planning, tool use, and feedback.

That distinction matters in XML projects because many requests are cross-cutting. “Rename this element everywhere” is not a text replacement. It can involve instance documents, schemas, stylesheets, selectors, tests, documentation, and publishing scripts. “Restructure this DITA map” may require classification, reference updates, validation, and publishing checks. Text generation alone is not enough for such tasks.

A sensible progression starts with assisted authoring and explanation, then moves to context-aware completion and review, and only later to multi-step orchestration. Each step gives the assistant more responsibility. Each step also needs stronger checks.

3. Key Areas Where AI Accelerates XML Work

1. **AI-assisted code generation.** AI is useful for producing first drafts of repetitive or well-understood XML artifacts. A developer can describe a mapping, a validation rule, or a documentation pattern and get a starting point for XSLT, Schematron, XSD, XPath, XQuery, or prose. The value is not that the result is final, but that it saves the initial blank-page work.
2. **Context-aware completion and explanation.** Traditional completion is driven mainly by syntax and schema constraints. AI can also take into account nearby code, naming patterns, active namespaces, and the likely intent of the current edit. In XSLT and Schematron, where the meaning often depends on surrounding context, a short explanation can be as useful as the suggested code itself.
3. **Review and refactoring support.** AI can inspect existing artifacts and point out duplicated rules, inconsistent naming, brittle XPath expressions, unclear templates, or missing documentation. It should be used to identify and draft candidate changes. For structural changes across a project, XML-aware refactoring and validation tools should still do the reliable work.
4. **Automation of project-specific tasks.** Many teams repeatedly normalize metadata, update namespaces, generate documentation, synchronize related files, convert semi-structured content, or prepare migration notes. AI can help users describe the intended operation in natural language and can help draft the scripts, queries, or repeatable actions that implement it.
5. **Agentic support and orchestration.** Some environments now allow an assistant to split a larger request into steps, choose tools, inspect intermediate results, and retry when validation fails. This is promising for migrations, reference updates, generated artifacts, and combined review-refactor-test cycles, provided the workflow exposes checkpoints and does not hide changes from the user.
6. **Onboarding and knowledge transfer.** XML projects often contain years of accumulated schemas, transformations, publishing rules, and local conventions. AI can help a new team member understand what a template does, why a rule exists, or how a map is organized. Those explanations still need to be checked against the implementation, but they can reduce the first barrier to entry.

4. Practical Scenarios

The following scenarios move from simple drafting to more coordinated workflows. They are not meant to describe fully autonomous systems. They show

places where AI can usefully share the work with XML-aware tools and with the developer.

1. **AI-assisted code generation.** Suppose a developer has sample XML and a desired HTML output. Asking an assistant for an initial XSLT stylesheet can be a good way to start. The generated stylesheet may map elements into a table, copy values into headings, or create links in the expected structure. The developer must still check the XPath expressions, output method, namespace handling, whitespace, and edge cases, but the first implementation pass is faster.

   The same pattern applies to validation rules. If an editorial rule says that every `chapter` needs a non-empty `title`, an assistant can draft a Schematron assertion. That draft is useful only after the namespace bindings, context expression, whitespace handling, and diagnostic wording have been checked against project conventions.

2. **Context-aware completion and explanation.** Consider an XSLT stylesheet that already contains templates for rendering a book. When the user starts a new template for a table cell or conditional image output, the assistant can inspect the existing templates and suggest code that follows the same style. If the editor also knows the source document used by the transformation scenario, the assistant can propose XPath expressions based on real instance data rather than generic placeholders.

   This is different from ordinary autocomplete. The suggestion is based not only on syntax, but also on the surrounding stylesheet and the apparent purpose of the edit. A good assistant should also be able to explain the suggestion, which helps the user decide whether to accept it.

3. **Review, refactoring, and knowledge transfer.** A stylesheet may contain separate logic for `order` and `Order`, or for `customer/name` and `CustomerName`, because different source systems use different conventions. An assistant can notice the inconsistency, explain why the current XPath is fragile, and suggest a more tolerant approach, for example by comparing `lower-case(local-name())` values in carefully limited places.

   The generated fix is only part of the value. The assistant can also draft a short project note: which source systems use which names, why the stylesheet handles them together, and what convention future templates should follow. That kind of documentation is often missing from long-lived XML projects.

4. **Automation of project-specific tasks.** In a publishing workflow, an XSLT transformation might call an AI-backed extension function to propose alternate text for images that lack accessibility metadata. XSLT remains responsible for traversing the documents, deciding where alternate text is needed, and inserting the result in the correct structure. AI contributes the language-oriented judgment that would be difficult to encode as rules.

Similar candidates include summaries, migration notes, short descriptions for generated catalogs, and first-pass conversion from semi-structured input into XML. In each case, the surrounding workflow should still validate the structure and leave the result open for review.

5. **Agentic support and orchestration.** An element rename is a simple request with a complicated implementation. A careful agent first discovers where the element is used. It then updates instance documents, schemas, XSLT stylesheets, CSS selectors, JavaScript or TypeScript DOM queries, tests, and documentation as needed. After editing, it validates the affected artifacts and reports both the changes made and any places it could not handle automatically.

   DITA map restructuring is another scenario. Starting from a flat map, an agent can classify topics as concept, task, reference, or troubleshooting material, suggest a hierarchy, rebuild the map as a `bookmap`, update `xref`, `conref`, and key relationships, validate the result, and run publishing checks. The useful point is not that the agent replaces an information architect. It is that it can perform much of the mechanical work while leaving the proposed structure visible for review.

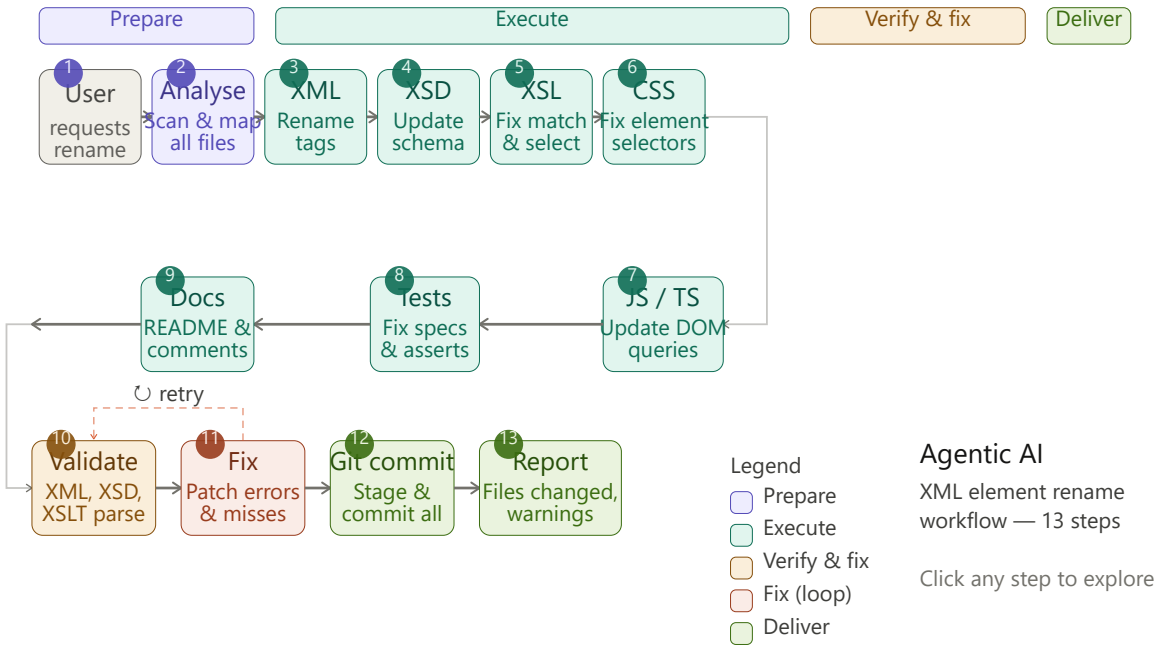


Figure 1. Agentic XML Element Rename Workflow

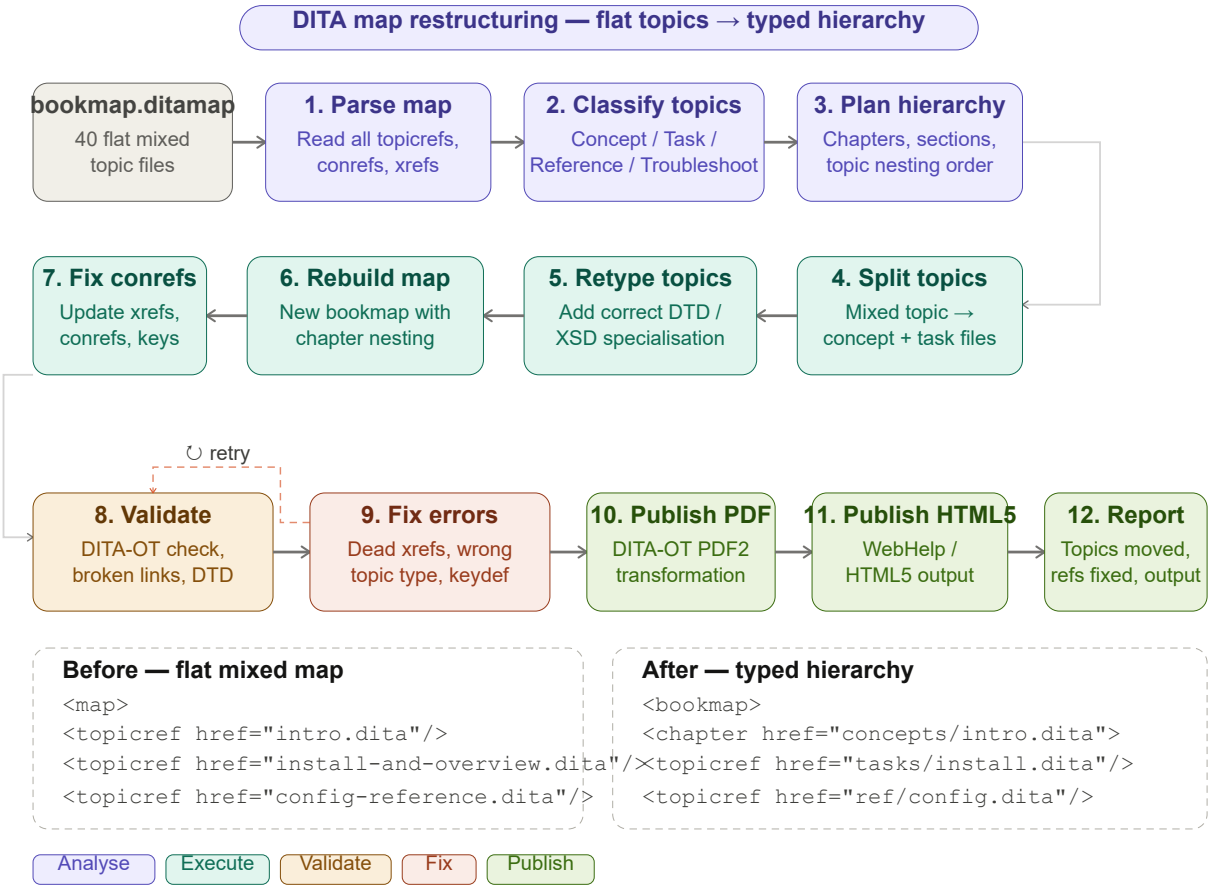


Figure 2. Agentic DITA Map Restructuring Workflow

5. A Practical Four-Stage Feedback Loop

The most reliable AI-assisted XML workflows have a small feedback loop built into them. The loop keeps the model's contribution visible and keeps the verifiable work in the hands of tools designed for XML.

1. **Gather context.** The system collects relevant information from the current document, schemas, stylesheets, validation rules, instance data, and nearby project files. Without this context, the model has to guess.

2. **Generate a draft.** The model proposes an explanation, query, transformation, rule, summary, or action plan. This is the uncertain part of the workflow and should be treated as a candidate result.

3. **Run deterministic checks.** XML tools validate the structure, evaluate XPath expressions, run transformations, apply refactorings, execute tests, and inspect references. Anything that can be checked mechanically should be checked mechanically.

4. **Review the result.** The user accepts, edits, or rejects the output. Human review is especially important for business meaning, publishing behavior, terminology, and local conventions.

The loop often runs more than once. Validation may expose a schema error. A transformation test may fail. A link check may show that one reference was missed. In an agentic workflow, the assistant can patch the result and try again, but the retries should remain bounded and visible.

XML validity is only one check among many. Real projects may also need Schematron or BREX validation, XSpec or regression tests, link integrity checks, terminology review, completeness checks, localization checks, publishing checks, Git diff review, and performance checks for large transformations. A document can be schema-valid and still be wrong for the workflow that uses it.

6. Integration Architecture: Bringing AI into XML Workflows

A conservative architecture works best. AI helps interpret requests, draft artifacts, and plan changes. XML-aware tools remain responsible for operations that must be exact: validation, refactoring, querying, testing, project-wide updates, and publishing checks. This separation is not a lack of ambition. It is a way to use AI where it is strong while protecting the parts of the workflow where precision matters most.

AI can enter an XML development environment in several ways:

- **Cloud APIs:** Hosted models are easy to try and often provide strong general capabilities for generation, explanation, review, and summarization. They require careful governance when the documents, schemas, or business rules are sensitive.
- **Local LLMs:** Local deployment can be preferable when privacy, compliance, or customer restrictions prevent sending content outside the organization. The tradeoff is that model quality, setup effort, and operating cost may vary.
- **IDE and editor integration:** XML editors already know about the active document, validation scenarios, schemas, transformations, and project resources. AI assistance is more useful when it can work inside that context instead of relying on pasted fragments.
- **Pipeline and service integration:** AI-backed services can support migration, classification, review, documentation, or enrichment tasks inside larger publishing and validation pipelines, especially when the work spans many files.

The right choice depends on the task, the required context, privacy constraints, and operational control. Quick explanations may work well with a cloud service. Sensitive customer content may require a local model. Project-wide refactoring benefits from editor or pipeline integration because the assistant needs access to the files, validation rules, and tests that define correctness.

In practice, the workflow design matters more than the model choice alone. Useful systems retrieve the relevant project context, constrain the prompt, make edits visible, log interactions when needed, and require approval at the right points. AI output should pass through the same gates as any other XML change: schema validation, Schematron validation, transformation checks, tests, and publishing verification.

Editor-centered integration also improves traceability. The prompt can be grounded in named resources, proposed edits can be shown as reviewable changes, and validation messages can be attached to the result immediately. For regulated or long-lived XML projects, that traceability is often more valuable than raw generation speed.

7. Tools and Skills for XML-aware AI Agents

An AI agent becomes useful for XML only when it can work with the resources that XML developers actually use. It needs to read XML files, schemas, stylesheets, maps, and project metadata. It needs to search for elements, attributes, keys, references, and naming patterns. It needs safe ways to edit, rename, move, and organize files. It also needs access to validation, transformations, tests, and sometimes version control or CI/CD systems.

Reusable **skills** are equally important. A skill is a set of task-specific instructions that tells the assistant how to work in a domain. In XML projects, useful skills might cover XSLT authoring and review, Schematron rule writing, XSD change analysis, XSpec testing, DITA map and topic operations, S1000D publication rules, or translation workflows that preserve markup.

Skills reduce guesswork. They capture conventions, examples, preferred tool usage, naming rules, and review expectations. Instead of starting from a generic prompt every time, the assistant can follow a stable pattern for recurring work such as validation, refactoring, migration, or documentation. In XML projects, that consistency matters because small deviations from local conventions can produce large downstream effects.

8. Challenges and Limitations

AI can make XML work faster and more approachable, but the risks are real enough that teams need disciplined usage.

- **Limited project context:** The assistant may miss dependencies between XML documents, schemas, stylesheets, maps, reused content, and publishing logic.
- **Confident but incorrect output:** Generated XML or code can look right while selecting the wrong nodes, using the wrong namespace, or covering only the sample case.

- **Validation is not enough:** A document may be schema-valid and still violate business rules, editorial conventions, accessibility requirements, or publishing expectations.
- **Incomplete changes:** A refactoring may update one artifact and miss related files, references, selectors, tests, or downstream transformations.
- **Security and privacy risks:** XML content may contain customer data, business rules, intellectual property, or regulated information.
- **Explainability and maintainability:** Teams need to understand why a rule or transformation exists, especially in systems that will be maintained for years.

Common failure modes include invented namespace prefixes, wrong namespace URIs, XPath expressions evaluated from the wrong context, stylesheets that handle only part of the input, refactorings that miss references, business-invalid but schema-valid documents, and broken IDs, keys, includes, or reused content. These problems are not reasons to avoid AI. They are reasons to keep validation and review close to every generated result.

The practical conclusion is simple: AI should accelerate XML expertise, not replace it. The more important the workflow, the more visible the checks must be.

9. Best Practices for Adoption

Teams evaluating AI for XML work usually get better results by starting small. Low-risk uses include explanation, documentation drafting, review assistance, sample data generation, and first-pass code generation for later review. Once those workflows are understood, teams can move toward migration support, rule generation, and AI-assisted editing tied directly to validation and refactoring.

Several practices help consistently:

- Use AI for acceleration, not authority.
- Provide project context instead of relying on generic prompts.
- Validate every generated artifact.
- Use XML-aware refactoring for structural changes.
- Keep people in the approval loop.
- Define checkpoints for agentic workflows.
- Verify each step with deterministic XML tooling.

These recommendations are cautious by design. XML projects often support publishing, compliance, interchange, or long-lived documentation systems. In those settings, correctness matters more than novelty. AI adoption succeeds when it removes friction without weakening the guarantees already provided by schemas, Schematron, tests, publishing checks, and expert review.

10. Conclusion

AI is becoming a practical part of XML development, especially when it is available inside the editor where the real work happens. It can help teams draft code, understand older assets, generate validation logic, review content, and handle repetitive tasks. Its value is greatest when it is combined with project context and with tools that can verify the result.

The strongest uses are drafting, explaining, reviewing, and planning. Agentic workflows extend that usefulness by coordinating tools and multi-step operations, but they also require visible checkpoints, deterministic verification, and human approval. AI should be one component in the XML engineering system, not a substitute for it.

For XML practitioners, the best stance is neither hype nor refusal. Use AI where it reduces friction. Keep relying on schemas, Schematron, tests, publishing checks, and expert review where correctness matters. In that model, AI does not replace XML expertise; it gives experts more leverage.

Publisher case study for XML-enabled quality-control techniques for journal metadata, pagination, and equations

M. Scott Dineen
Optica Publishing Group
<sdineen@optica.org>

Abstract

Optica Publishing Group (OPG) is an innovative society publisher of 20 journals in the field of optics and photonics and the founder of one of the world's first online-only journals, Optics Express. An early adopter of XML and MathML, OPG has a veteran publishing staff with a reputation for technology innovation. In this paper, I share three areas where OPG staff recently used XML-aware methods in creative ways to address production pain points that can have significant reputational or financial stakes: journal article metadata, PDF pagination, and math notation. This short case study will challenge readers to pursue creative solutions for critical gaps in their own procedures.

1. Journal Metadata

Along with peer review, a core service journal publishers provide is the collection and curation of metadata for downstream processes. Publishers not only register metadata with Crossref and other indexers, they also rely on metadata such as funder information and license details to fulfill a growing list of obligations to authors, institutions, and the wider community.

One such obligation that relies on metadata is the Read and Publish agreement, in which an institution pays a publisher for both subscription access and open-access publishing fees for authors affiliated with that institution. To make such an arrangement work, when authors submit to a journal, the publisher must correctly identify the author's institution so that authors receive their entitlements and aren't incorrectly billed for open-access fees. Identifying an author's affiliation may sound trivial, but for a large international publisher with a substantial journal portfolio, the task presents real challenges, and those challenges are compounded by how the metadata is collected in the first place.

When authors submit journal manuscripts, much of their metadata—author names, funders, affiliations, and so on—is captured screen by screen through a submission system, sometimes aided by type-aheads and other interface tools.

This data is sometimes simply incorrect and, to make things worse, what the author enters in the submission system doesn't always match what appears in the manuscript itself. Publishers attempt to reconcile the two sources by applying business rules that designate which source is the authority for which piece of data, but inevitably, some mismatches slip through into late-stage production, and the consequences of publishing the wrong data can be severe. Consider the reputational risk of leaving an author's name off a published paper or of omitting a research funder (and the specific publishing license that funder may require).

To help catch these issues prior to publication, OPG built a metadata comparison report that XML production staff can launch conveniently from their oXygen XML Editor software while working on the XML version of an article. As part of the process, data from the submission system is written out to a separate XML file, and an XSLT script generates a color-coded HTML report comparing information from the article and database dump from the submission system side by side. The report design balances automated detection with human review for items that require human judgement, and the process works in tandem with a library of Schematron business rules. If staff add a missing funder to the XML manuscript, for example, Schematron rules will alert them if that funder also requires a special license (see Fig. 1).

2. Pagination Fidelity

Pagination errors are among the worst offences to appear in a published article. Gaps in pagination or overlapping page numbers can affect an entire journal issue and create lasting problems for citation and indexing. Because the consequences are so severe, publishers go to great lengths to prevent pagination errors, but a number of factors still introduce risk, including special feature journal issues with alternate (e.g., alphanumeric) pagination and late-stage production changes that alter an article's page count.

At the production stage, a key technical challenge with pagination is keeping the page information in sync between the XML manuscript and the composed PDF. When late-stage corrections shift the layout of a PDF, the new page numbers must propagate back to the XML manuscript and to the database record. Ensuring that this happens consistently can be tricky, and a single missed update can result in broken Crossref deposits, mismatched indexing, and citation links that point to the wrong page.

To address this risk, OPG developed a Schematron-based method for keeping the PDF and XML manuscript sources aligned at a natural point where staff can take action. Each time an article is composed to PDF, a separate XML file is generated from the PDF's XMP metadata, capturing the page information for that article. Schematron rules running in oXygen Editor then compare the pagination in the article XML against the values in the XMP file and flag any discrepancies

Manuscript Number OL-578473

RightsLink Status: NONE

Total PDF Pages : 4 (please check against the actual PDF)

Composed PDF	JATS XML	Match?
Total Page Count (cover page discounted)		
4	4	total-pages-Match
First Page		
7255	7255	first-pages-Match
Last Page		
7258	7258	last-pages-Match
Prism Starter	**JATS XML**	**Match?**
Authors		
Author Count: 6	Author Count: 6	count-Match
Li, W enwen	Li, W enwen	name-Match
pan, t uqiang | 0000-0002-1376-2106	Pan, T uqiang | 0000-0002-1376-2106	name-Match | orcid-Match
Qin, Y uwen | 0000-0001-9879-1514	Qin, Y uwen | 0000-0001-9879-1514	name-Match | orcid-Match
Xu, Y i | 0000-0003-1679-3271 | *corresponding	Xu, Y i | 0000-0003-1679-3271 | *corresponding	name-Match | orcid-Match
Yang, J iayi	Yang, J iayi	name-Match
Zhang, Z hanyuan | 0000-0002-5689-9145	Zhan, Z hanyuan | 0000-0002-5689-9145	name-Mismatch | orcid-Match
ROR Organization		
Guangdong University of Technology https://ror.org/04azbjn80		If there is an ROR ID in the starter file, ensure that the ROR ID is in place **and** with the correct institution in the JATS file. Note that the institution names do not need to be an exact match.
Funding		
Guangdong Introducing Innovative, Entrepreneurial Teams of "The Pearl River Talent Recruitment Program" (2019ZT08X340; 2021ZT09X044) National Natural Science Foundation of China 10.13039/501100001809 (62222505; 62335005)	National Science Foundation (62222505; 62335005)	funder-name-Mismatch

Figure 1. Metadata report comparing information from the submission system database with that in the XML manuscript.

for staff to resolve. The technique has worked well and has eliminated published page overlaps in OPG journals since its introduction (see Fig. 2).

Engine name: ISO Schematron

Description: ERROR [J_PUBL:MPUB190] article 597318: expected the 4th segment of <license-p> to be "10", found "04"

Description: ALERT [J_PUBL:OPGH270] article 597318: <lpage> [2801] in "597318.xml" does not match [2796] in "597318_page.xml" - please verify

Description: ALERT [J_PUBL:OPGH270] article 597318: the total number of pages [10] in "597318.xml" does not match <totalpage> [5] in "597318_page.xml" - please verify

Figure 2. Pagination mismatch between PDF and XML metadata identified with Schematron.

3. Equation Accuracy

Arguably the most challenging content to produce in physics papers is mathematics. Authors submit math equations in a variety of formats such as MathType, Microsoft OMML, and LaTeX, and the publisher must then normalize the formats to a single standard for production and composition. As was mentioned, OPG was an early adopter of MathML (Mathematical Markup Language), and we worked closely with the American Institute of Physics in the early 2000s to improve and integrate Design Science tools (such as MathFlow) for editing and rendering MathML in an XML pipeline.

For nearly two decades, OPG used MathML to encode equations for production. The approach worked but required a patchwork of proprietary tools for editing, PDF composition, and HTML display. Each tool interpreted the MathML slightly differently, and over time we found ourselves tuning the markup to suit the idiosyncrasies of a proprietary composition system. Inconsistent math rendering across tools was one problem. Another was that MathML is simply too difficult for production staff to edit efficiently. WYSIWYG tools never gave us sufficient control and could rewrite the markup in unexpected ways, while editing raw MathML was preferable in principle but daunting in practice given how verbose the format is.

To address both problems, we re-engineered our production process to normalize equations to LaTeX rather than MathML. This change offered several advantages. LaTeX math is by far easier to edit than MathML and provides more predictable results. Second, regarding rendering consistency, by the time we started the project, MathJax (an open-source JavaScript display engine supported by publishers worldwide) had become the *de facto* standard for rendering both MathML and LaTeX in the browser, so we adopted MathJax as the single rendering engine across our tool set. Finally, normalizing all math to LaTeX allowed us to move away from proprietary XML+MathML composition systems and simply convert the XML+LaTeX to a format that could be composed to high-quality output with LaTeX directly. Migrating to LaTeX math sped up our processes, reduced author corrections at the page-proof stage, and brought down our vendor costs as the workflow became more automated. It also made authors who submit in LaTeX noticeably happier, since we no longer had to disturb their carefully prepared math notation.

With the change to LaTeX math, however, we did encounter one unexpected obstacle. For more than a decade, OPG has used oXygen XML Editor desktop software for production and has made heavy investments in customizations. At the time we changed our workflow, oXygen XML Editor did not have an adequate LaTeX math editing plug-in. We would have to develop our own tool for editing the LaTeX and rendering it with MathJax within oXygen. Fortunately our need coincided with the AI revolution. Building such a plug-in would have

been hard to justify even a few years before, but by early 2024, with help from ChatGPT, even a few "vibe coders" in our publishing group were able to produce a stable and production-ready tool that previously would have required either a vendor contract or a dedicated developer ??? (see Fig. 3).

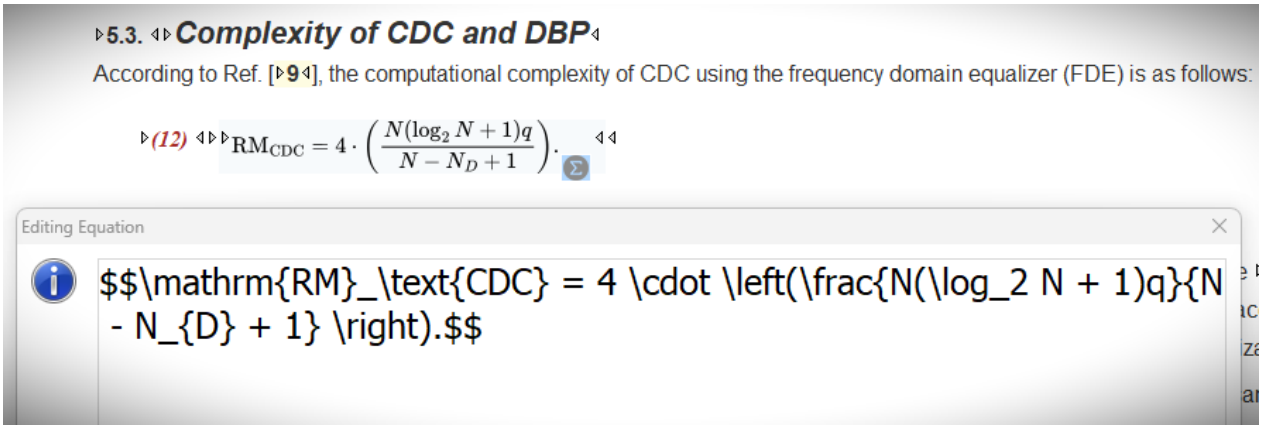


Figure 3. OPG's LaTeX plug-in for Oxygen XML Editor desktop software. A MathJax server receives the LaTeX and returns a rendered SVG file to the preview (top portion below) of the editing environment.

Looking back, we've asked ourselves why we didn't migrate from MathML to LaTeX math sooner. One reason is that we had convinced ourselves that an XML workflow should use XML conventions wherever possible, including MathML for math. It turned out, though, that embedding LaTeX within the XML manuscript provided the best of both worlds. In addition to enjoying the benefits described above for using LaTeX notation, within an XML workflow we were able to leverage tools like Schematron to detect unwanted patterns in the math (such as the negative space macro `\!`, which some authors love for fine-tuning, but causes problems in MathJax). We were also able to store alternative versions of equations conveniently in the XML using the JATS (Journal Article Tag Suite) XML option for `alternatives` (see Fig. 4). Storing multiple versions of the math was necessary, because to meet accessibility obligations, we had to generate MathML from the LaTeX at the very end of the publishing process (via MathJax) and store it alongside the LaTeX as a JATS alternative, as shown in the figure, for use in contexts where MathML is required.

4. Conclusion

The three projects described in this paper—metadata reconciliation, pagination validation, and the move from MathML to LaTeX—address very different production problems, but they share a common pattern. In each case, OPG staff identified a workflow gap with real reputational, financial, or compliance stakes and closed it by combining established XML technologies (XSLT, Schematron,

```
<inline-formula>
    <alternatives>
        <tex-math notation="LaTeX" version="MathJax"
            >$\frac{\textrm{P}}{\textrm{A}}$</tex-math>
        <mml:math display="inline">
            <mml:mfrac>
                <mml:mrow class="MJX-TeXAtom-ORD">
                    <mml:mtext>P</mml:mtext>
                </mml:mrow>
                <mml:mrow class="MJX-TeXAtom-ORD">
                    <mml:mtext>A</mml:mtext>
                </mml:mrow>
            </mml:mfrac>
        </mml:math>
    </alternatives>
</inline-formula>
```

Figure 4. MathML stored alongside LaTeX markup in the XML manuscript.

JATS, MathJax) with judgment about where automated checks were sufficient and where human review remained essential. None of the underlying tools is exotic, and none of the problems is unique to OPG. Publishers everywhere are under growing pressure to meet accessibility requirements, funder mandates, and a range of other commitments, most of them driven by article metadata, and nearly all of them carrying consequences if missed.

What may be transferable across publishers is the approach rather than any specific implementation. Look closely at where errors actually originate in the workflow, and ask whether an XML-aware check applied at the right point can surface them before they reach the published record. Critical gaps in production processes are rarely closed by a single sweeping change but more commonly by small, well-targeted changes built on tools and standards that staff already understand.

References

[1] https://github.com/OpticaPublishingGroup/LaTeX-in-oXygen-Editor .

XTH – An Implementation Agnostic Test Suite Runner

Adam Retter
Evolved Binary
<adam@evolvedbinary.com>

Abstract

An important aspect of the W3C's (World Wide Web Consortium's) process when developing new standards includes ensuring that the standards are realistically implementable. To ensure that an implementation is operating in accordance with the standard, the W3C typically also produces and publishes one or more 'Test Suites'.

Our contributions include, firstly, an analysis for the purpose of understanding the similarities and differences of the Test Suites produced for the W3C XML (eXtensible Markup Language) Query Language (i.e. XQuery), and the W3C XSLT (eXtensible Stylesheet Language for Transformations) standards.

Secondly, an examination, focused on implementation and compliance challenges, of prior efforts by vendors implementing these standards to execute these Test Suites and assert their compliance.

Finally, we contribute XTH (X Test Harness) to the XML community, a freely available and modifiable software tool for executing the Test Suites against vendor implementations of the standards.

Keywords: W3C, XQuery, XSLT, XML, Standards, Testing, Compliance

1. Introduction

When the W3C works toward new standardisation efforts for a technology, the first step is to form a WG (Working Group) centred on that technology within the W3C [1]. Development of standards within a WG follow a process prescribed by the W3C. Each new standard moves through four phases until it is recommended by the W3C:

1. WD (Working Draft)
   The specification is developed during this phase, feedback gathered and addressed; there may be many distinct draft versions.

2. CR (Candidate Recommendation)

The specification is considered stable, and it must now be proven that it is accessible and realistically implementable.

3. PR (Proposed Recommendation)
   An implementation report is produced and must be reviewed and approved by the W3C Advisory Committee.
4. REC (Recommendation)
   The specification is now considered a web-standard, and enters a maintenance only phase.

During the CR phase, a WG must prove that each of their standard specifications is realistically implementable by vendors[1]. One such mechanism available to a WG for this purpose is for them to develop and publish a set of tests, a 'Test Suite'. For a WG to progress their development of a standard from the CR to PR phase, it is common for them to be required to demonstrate a minimum of two interoperable implementations [2] of that standard. Such a Test Suite can be used to measure compliance of a vendor's product, and demonstrate correctness in implementation and interoperability.

After the CR phase or even the REC phase, a Test Suite for a standard can still continue to deliver a great deal of value. For the WG, the Test Suite allows new vendors to arrive in future and correctly implement the standard; more vendors, means wider standards adoption. For a vendor, it initially provides validation that they have correctly implemented the standard, and in future it allows them to change their product whilst validating against introducing any regressions.

The Test Suites produced by a WG are typically not in the form of directly executable computer code. Rather, they tend to define a number of distinct tests, that may be grouped into different categories. Each of the defined tests may:

- Describe how to setup some sort of computational environment.
- Describe a number of dependencies (e.g. files) that are required by the test.
- Describe a computation or action(s) to be performed.
- Describe a number of assertions about, the results of the computation, or the environment post computation, that must hold true.

This paper focuses on the Test Suites produced by two W3C WGs:

1. XML (eXtensible Markup Language) Query Language (i.e. XQuery)
2. XSLT (eXtensible Stylesheet Language for Transformations)

These two WG were closely intertwined by virtue of their advances being underpinned by the same XML and XPath technologies, and through many individuals participating in both groups. Each of these two WG produced a number of Test Suites, and perhaps therefore unsurprisingly, expressed their Test Suites using

[1]We use the term 'vendor' to refer to a person or organisation, that commercially or freely, produces a software product that implements a standard.

very similar methods, languages, and technologies. We analyse the Test Suites in section: Analysis of XQuery and XSLT WG Test Suites.

Implementation of one or more of the W3C XQuery and/or XSLT standards by any vendor is a large technical undertaking; the XQuery 3.0 test suite alone contains more than 27,000 test cases. There is a further technical burden for any vendor who wishes to ensure that they are compliant with a standard. They must also implement additional software, a 'Test Harness', to process the Test Suites published for a specific standard by the WG. Worse yet, if the vendor is implementing multiple standards, and those standards express their Test Suite(s) in different manners, they may be forced to develop multiple Test Harnesses. We examine existing vendor test harnesses in section: Analysis of Vendor XQuery and XSLT Test Harnesses.

Vendors often like to publicise their compliance with a standard, by stating the number of tests within a Test Suite that they pass (i.e. assertions after computation hold true). Therefore, from a standards compliance perspective, it would be improper and undesirable for a test harness to report that a test from a Test Suite is incorrectly passing or failing when it is fact not. Understandably then, development of such test harnesses requires great vigilance and attention to detail, and correctness of the Test Harness software is paramount, perhaps more so that the implementation of the standard itself.

We have developed and are contributing XTH (X Test Harness) to the XML community. XTH is freely available and modifiable software that can parse, dispatch for execution, and report on the results of all tests in the Test Suites produced by the W3C XQuery and XSLT WGs. XTH offers a number of advantages for each audience over the contemporary Test Harness approaches:

- Working Group
  Verifying compliance with a standard's Test Suite now requires much less work by a vendor, which should mean faster/wider standards adoption.
- Vendor
  No need to implement a complex custom Test Harness. Instead, implement a small product specific connector.
- Developer
  Easier to prototype new features and changes in your product and test compliance.
- Researcher
  Employ the scientific experimental method openly to reproduce and/or confirm/refute claims of standards compliance.
- User
  Easily check standards compliance of available implementations to determine if they match your requirements.

We detail the architecture of the XTH software in section: Introduction to XTH (X Test Harness).

2. Analysis of XQuery and XSLT WG Test Suites

The following standards and test suites were produced either separately or in collaboration by the W3C XQuery and XSLT WG[2]

XQuery WG	
Standard	**Test Suite**
XQuery 1.0 [3]	XQTS 1.0.3 [4]
XQuery Update Facility 1.0 [5]	XQUTS 1.0.1 [6]
XQuery 3.0 [7]	QT3 1.0 [8]
XQuery Update Facility 3.0 [9]	None available.
XQuery 3.1 [10]	QT3 Tests [11]

XSLT WG	
Standard	**Test Suite**
XPath 1.0[a] [12]	Indirectly through the XSL 1.0 Test Suite.
XSL 1.0 [13]	XSL 1.0 Test Suite [14]
XSL 1.1	XSL 1.1s Test Suite [15]
XSLT 1.0 [16]	XSLT Test Suite 04 [17]
XSLT 2.0 [18]	Not publicly available.
XSLT 3.0 [19]	XSLT 3.0 Test Suite [13]

[a]Jointly produced in collaboration with the XML Linking Working Group.

[2]Prior to January 2011, the XSLT WG was known as the XSL WG. For the purposes of simplification in this paper we consistently use the name XSLT WG.

Joint XQuery and XSLT WGs	
Standard	**Test Suite**
XPath 2.0 [20]	Indirectly through the XQTS 1.0.3, and the private XSLT 2.0 test suite.
XQuery 1.0 and XPath 2.0 Data Model [21]	
XQuery 1.0 and XPath 2.0 Functions and Operators [22]	
XSLT 2.0 and XQuery 1.0 Serialization [23]	
XQuery and XPath Full Text 1.0 [24]	XQFTTS 1.0.4 [25]
XPath 3.0 [26]	Indirectly through the QT3 1.0, and the XSLT 3.0 Test Suite.
XQuery and XPath Data Model 3.0 [27]	
XPath and XQuery Function and Operators 3.0 [28]	
XSLT and XQuery Serialization 3.0 [29]	
XQuery and XPath Full Text 3.0 [30]	None available.
XPath 3.1 [31]	Indirectly through the QT3 Tests, and the XSLT 3.0 Test Suite.
XQuery and XPath Data Model 3.1 [32]	
XPath and XQuery Function and Operators 3.1 [33]	
XSLT and XQuery Serialization 3.1 [34]	

In addition, since 2019 when the W3C's XQuery WG and XSLT WGs were closed, there has been a CG (Community Group) [35] effort, to develop newer standards for XPath, XQuery, and XSLT. No final standards have yet been published, but Test Suites for proposed XPath and XQuery 4.0 [36], and XSLT 4.0 standards are being [37] developed publicly.

In total there are 11 Test Suites available from the XQuery and XSLT WGs. We now examine them chronologically.

The earliest two test suites, XSL 1.0 Test Suite, and XSL 1.1 Test Suite, are specified using the same simple XML grammar with an accompany DTD (Document Type Definition). Each test definition requires an XML Input File and an XSL Input File as input, and links to an Image or PDF file that shows an example visual representation of what the output from the XSL Formatter should produce. The results of the tests themselves cannot be asserted computationally, as each generates a visual output that has to be compare 'by eye' against the provided example output. As XSL was superseded by splitting the language into two parts:

XSL-FO (XSL Formatting Objects) and XSLT (XSL Transformations), there is little value in concerning ourselves further with these specific test suites.

The XSLT Test Suite 04 was actually developed outside of the W3C by a separate standards organisation called OASIS (The Organization for the Advancement of Structured Information Standards). This test suite is specified using a different simple XML grammar, that is centered around the concept of a "Catalog" of tests. Unfortunately, neither a DTD or XML Schema is provided. The main file is named `catalog.xml`, and defines all tests in the test suite. Each test definition requires takes as input, one or more XML and XSLT files, and links to an XML file that the output of the test must match. A comparison is performed between the output of the test execution and the supplied output file by means of Canonical XML 1.0 [38]. Through this mechanism, unlike the prior test suites, the results of the tests may be computed automatically.

XQTS 1.0.3 test suite (and its prior revisions) build upon the Catalog concept introduced by XSLT Test Suite 04. They utilise a more extensive XML grammar, and an XML Schema (`XQTSCatalog.xsd`) is provided to define the format of the catalog. The catalog (file `XQTSCatalog.xml`) defines the test suite, which is broken up into groups of tests, and individual test cases. Each Test may define zero or more inputs, an environment, an XQuery to execute, and either (a) expected results in XML or Text format (canonicalisation must be applied before comparison), (b) an expected error, or (c) that a human inspection is required to ascertain correctness or the result. In practice only 56 out of the almost 20,000 test cases require a human to inspect the results, meaning that for the most part, the results of the tests can be computed automatically.

The XQUTS 1.0.1 test suite (and its prior revisions) utilise largely the same grammar (`XQUTSCatalog.xml`) as the XQTS albeit in a different namespace. XQUS also adds additional grammar elements to each test case definition that allow describe a sequence of state changes that are performed by an XQuery.

The XQFTTS 1.0.4 test suite (and its prior revisions), like XQUS, remain close to the grammar of XQTS. Like XQUS it utilises its own distinct namespace. The only substantial change in the XQFFTS grammar (`XQFTTSCatalog.xsd`) over the XQTS grammar, is the addition of 3 new types of sources, each required by the full-text standard: stop words, thesaurus, and stemming dictionary.

The grammar of the QT3 1.0 test suite (`catalog-schema.xsd`) whilst conceptually similar to the XQTS 1.0.3 test suite, should be viewed as a major redesign. Like XQFTTS and XQUS is utilises its own distinct namespace. Whilst it keeps the Catalog centric approach, the catalog is now broken into multiple files, where each sub-file is a set of tests, defining multiple test cases. For the inputs to tests, dependencies and environments can now be setup globally (if desired) and reused across multiple tests. In addition the mechanism for determining the result of a test execution has been greatly expanding, alongside canonical comparison of a test's results against a known XML document, there is now an

entire DSL (Domain Specific Language) for making assertions about the results of a test. All tests in QT3 can have their results computed automatically.

The XSLT 3.0 Test Suite directly builds upon the grammar of QT3 1.0 test suite, as it states in its own grammar (`admin/test-catalog.xml`):

> *whose design is based closely on the QT3 test suite used for XPath 3.0 and XQuery 3.0 testing. The schema is not identical to the QT3 schema (some improvements have been made, and some adaptations to the different needs of XSLT) but the design will be familiar to anyone with experience of QT3, and it is possible to build test drivers for both test suites using common code.*

The test suite utilises its own namespace, and mainly adds some constructs that are specific required for testing XSLT 3.0, such as support for packages, tunnelled parameters, initial functions/templates/modes, and streaming (posture and sweep). It also adds some new assertions to the assertion DSL, including: not, assert-message, assert-warning, and assert-posture-and-sweep.

The grammar of the QT3 Tests test suite (`catalog-schema.xsd`), and the QT4 Tests test suite (`catalog-schema.xsd`), share the same namespace as the QT3 1.0 test suite. The QT3 Tests test suite makes only minor enhancements over the QT3 1.0 grammar. The QT4 Tests test suite likewise adds few enhancements apart from a facility for describing filesystem mutations and potentially XQuery Updates.

Likewise, the grammar for the XSLT 4.0 Tests test suite, shares the same namespace as the XSLT 3.0 Tests test suite. It adds very few minor enhancements that are solely focused on testing XSLT 4.0 features.

From our analysis of the test suites produced by the XQuery and XSLT WGs, we can observe a great deal of similarity in their grammars. Excluding the now legacy XSL 1.0 and 1.1 test suites, there are 3 main classes of test suite grammars that we need concern ourselves with when testing vendors implementations:

1. XSLT Test Suite 04
2. XQTS like:
   - XQTS 1.0.3
   - XQUTS 1.0.1
   - XQFTTS 1.0.4
3. QT3 like:
   - QT3 1.0
   - XSLT 3.0 Test Suite
   - QT3 Tests
   - QT4 Tests
   - XSLT 4.0 Tests

Based on our analysis, we believe that it would be reasonable to postulate that a common grammar or data model could be developed that would allow us to

describe the definitions and requirements of all of these tests regardless of their original grammar. We have developed such a model within our XTH software described in section: Introduction to XTH (X Test Harness).

3. Analysis of Vendor XQuery and XSLT Test Harnesses

For a vendor implementing one or more of the standards published by the XQuery or XSLT WGs it has long been the case that they will need to develop their own test harnesses to execute the provided test suites and assert their compliance with the standard.

From within the XQTS 1.0.3 test suite, the file `Guidelines for Running the XML Query Test Suite.html` clearly states:

> *Implementers are expected to write their own test harness that implements the following tasks: [a] Read test cases from the catalog, apply customization if applicable [...], [b] Execute tests, using source files specified in the catalog, [c] Use appropriate comparator to match result, [and d,] Produce categorization of test result (pass, fail etc., see below. Ideally, the test harness produces an XML file containing all test results in the format shown below, that can be sent to the working group.*

We can gain an understanding of why the WGs position is that vendors must implement their own test harnesses from the the file `release_notes.html` within the XQUTS 1.0.1 test suite:

> *As there is no de jure or de facto API for implementations of XQuery, we are not able to provide a test harness to execute these tests. You will have to write your own test harnesses.*

Correctly implementing a test harness that can execute a test suite is a significant undertaking for any vendor. Whilst there are vendor agnostic general APIs for invoking XQuery and XSLT such as XQJ (XQuery API for Java) [39] and JAXP (Java API for XML Processing) [40] they do not provide the necessary level of control over the vendor's software required for an implementation to execute a test suite.

This unfortunately has forced each vendor to implement their own test harnesses, and resulted overall in a great deal of duplicated work between vendors. Whilst some aspects of setup and execution of a test can only be vendor product specific, we would argue that other aspects such as *[a] read test cases from the catalog*, *[c] use appropriate comparator to match result*, and *[d] Produce categorization of test result* could be generalised, developed once as a set of common modules and utilised by all vendors. This is the approach that we have taken in XTH which is discussed in section: Introduction to XTH (X Test Harness). Such a set of common modules, can benefit from being open, and expertise shared to ensure correct implementation.

Across all of the Test Suites that we analyzed in section: Analysis of XQuery and XSLT WG Test Suites, we have identified 34 vendors that executed test suites and reported their results. Of those 34 vendors, whilst not conclusive, we have only been able to identify 6 vendors that, have published the source code of their test harnesses. From the point of the experimental scientific method, this lack of transparency is unfortunate, as it means that neither researchers not users can attempt to reproduce the findings in the vendor's compliance statements. Even when vendors have published the source code of their test harnesses, they have often failed to publish the specific configuration settings they applied to their product to achieve the results, this again hinders the ability to reproduce their results.

Published vendor Test Harness source code:

- BaseX [41] [42]
- Elemental [43] [44]
- eXist-db [45] [46] [47]
- RumbleDB [48]
- Saxon [49] [50]
- Zorba [51]

By examining the above test harnesses, we can see that each is complex, and that each vendor has taken a very different approach to implementing their Test Harnesses. Several vendors have developed more than one test hardness, with newer versions replacing older versions. From examining the history of these test harnesses, the results submitted and their subsequent updates, we can identify that like all software these test harnesses have also experience non-correctness issues at times. It is clear that some test harnesses have at points reported both false positive results and false negative results.

We argue that where possible, if vendors could reuse and contribute to a commons set of modules that make up a test harness, then Linus's Law should apply: *given enough eyeballs, all bugs are shallow* [52]; this is one of the guiding principles behind our proposal for XTH.

4. Introduction to XTH (X Test Harness)

We have developed XTH (X Test Harness) for the XML community. XTH is a freely available and modifiable software tool (with source code provided) for executing the Test Suites against vendor implementations of the XQuery and XSLT WG standards.

We developed XTH for the following reasons:

- Reduce the effort required by vendors to implement test harnesses.
- Avoid duplicated work across vendors.

- Reduce the amount of time required for standards groups to see adoptive implementations.
- Enable improved reporting of standards compliance for the benefit of users who wish to adopt standards.
- Multiple contributors from different vendors to XTH should reduce bugs, and therefore less reporting of false-positive or false-negative results.
- To enable researchers and users to easily reproduce and confirm/refute the findings of vendor compliance statements.

XTH is vendor agnostic and uses a modular pluggable architecture. Each vendor need only add a small connector module for their product. XTH provides common modules for parsing the test suites, creating an in-memory model of a test suite, scheduling execution of tests, and asserting and reporting on the results of test execution. XTH consists of the the following main components:

1. An in-memory model of a Test Suite. This is a generalised super-set of the models expressed in the XQuery and XSLT WG Test Suites. This model is defined as a set of Java interfaces and classes.
2. A Parser API and implementations thereof. Parsers may be added for different Test Suites. The parser reads the Test Suite definitions published by the WGs, and creates the (above) in-memory model.
3. An Execution Scheduler that decides how to dispatch the test cases from the Test Suite for execution to one or more Connectors. It also provides the facilities for asserting the results of an executed test.
4. A Connector API and implementations thereof. Connectors may be added for each vendor's product. Each connector is quite small and requires little more than the ability to execute a test (e.g. an XPath, XQuery, or XSLT) and receive the results.
5. A Reporting API and implementations thereof. Reporters receive results from an Execution Scheduler and format them for output.
6. A Command Line Interface (or Library) with configurable options that drives the entire process.

When a vendor wishes to implement a Test Harness with XTH, assuming that a suitable Parser for the Test Suite they are developing against exists, and that a Reporter exists for the results output format they desire, then they need only implement a small Connector for their product.

XTH is written in Java 25 (so that it may run on almost any platform), and has almost no external dependencies. It makes use of the Java Service Loader API to enable a developer to easily use new Parsers, Connectors, and Reporters.

We welcome contributions to XTH from both vendors and users alike. XTH is available from GitHub under a Fair Source license – https://github.com/evolvedbinary/xth.

5. Conclusion

We believe that we have made strong arguments for the case that XTH can reduce the effort required by vendors to implement Test Harnesses for the XQuery and XSLT WG standards, and that it can also increase the transparency and reproducibility of Test Suite results reporting.

At present, for XTH, apart from the common modules, we have contributed the following test suite, vendor, and reporting modules:

1. Parser Module for the QT3 Tests test suite.
2. Connector Module for the Saxon[3] HE/PE/EE XSLT and XQuery processors [53].
3. Reporter Module for Console reporting.
4. Reporter Module for OTR (Open Test Reporting) [54] event based XML format.

Our ideas for future expansion of this work include:

- Creation of an XML configuration grammar for XTH, to more easily enable shareable configurations for executing test suits against various vendor products.
- Publishing a matrix of results online that were generated by XTH against various combinations of test suites and vendor products.
- Providing a mechanism to support automation of results submission to the relevant WG party.
- Integrating XTH with Continuous Integration so that a matrix of test results is updated over time, and a change history of compliance is visible online.

6. Bibliography

Bibliography

[1] World Wide Web Consortium. *W3C Process Document*. 2025. https://www.w3.org/policies/process/ .

[2] World Wide Web Consortium. *Implementation Experience - W3C Process*. 2025. https://www.w3.org/poli cies/process/#implementation-experience[1] .

[3] W3C XQuery Working Group. *XQuery 1.0: An XML Query Language (Second Edition)*. 2010. https://www.w3.org/TR/xquery-10/ .

[3] A Connector Module for the Elemental XML Database has also been developed but is not yet ready for publication.

[1] https://www.w3.org/policies/process/#implementation-experience

[4] W3C XQuery Working Group. *XML Query Test Suite 1.0*. 2010. https://dev.w3.org/2 006/xquery-test-suite/PublicPagesStagingArea/[2] .

[5] W3C XQuery Working Group. *XQuery Update Facility 1.0*. 2011. https://www.w3.org/TR/xquery-update-10/ .

[6] W3C XQuery Working Group. *XQuery Update Facility Test Suite*. 2010. https://dev.w3.org/2007/xquery-up date-10-test-suite/[3] .

[7] W3C XQuery Working Group. *XQuery 3.0: An XML Query Language*. 2014. https://www.w3.org/TR/xquery-30/ .

[8] W3C XQuery Working Group. *XQuery/XPath/XSLT 3.* Test Suite*. 2014.

[9] W3C XQuery Working Group. *XQuery Update Facility 3.0*. 2017. https://www.w3.org/TR/xquery-update-30/ .

[10] W3C XQuery Working Group. *XQuery 3.1: An XML Query Language*. 2017. https://www.w3.org/TR/xquery-31/ .

[11] W3C XQuery Working Group. *qt3tests*. 2017. https://github.com/w3c/qt3tests .

[12] W3C XSLT Working Group. *XML Path Language (XPath) 1.0*. 1999. https://www.w3.org/TR/xpath-10/ .

[13] W3C XSLT Working Group. *XSLT 3.0 Test Suite*. 2017. https://github.com/w3c/xslt30-test .

[14] W3C XSL Working Group. *XSL 1.0 Test Suite*. 2001. https://www.w3.org/Style/XSL/TestSuite/ .

[15] W3C XSL Working Group. *XSL 1.1 Test Suite*. 2006. https://www.w3.org/Style/XSL/TestSuite1.1/ .

[16] W3C XSLT Working Group. *XSL Transformations (XSLT) Version 1.0*. 1999. https://www.w3.org/TR/xslt-10/ .

[17] OASIS XSLT Conformance Technical Committee. *Probably final test suite package*. 2005. https://lists-archive.oasis-open.org/archive/lists/xslt-conformance/months/200504/messages/1[4] .

[18] W3C XSLT Working Group. *XSL Transformations (XSLT) Version 2.0 (Second Edition)*. 2021. https://www.w3.org/TR/xslt20/ .

[19] W3C XSLT Working Group. *XSL Transformations (XSLT) Version 3.0*. 2017. https://www.w3.org/TR/xslt-30/ .

[2] https://dev.w3.org/2006/xquery-test-suite/PublicPagesStagingArea/
[3] https://dev.w3.org/2007/xquery-update-10-test-suite/
[4] https://lists-archive.oasis-open.org/archive/lists/xslt-conformance/months/200504/messa ges/1

[20] W3C XQuery and XSLT Working Groups. *XML Path Language (XPath) 2.0 (Second Edition)*. 2010. https://www.w3.org/TR/xpath20/ .

[21] W3C XQuery and XSLT Working Groups. *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. 2010. https://www.w3.org/TR/query-datamodel/ .

[22] W3C XQuery and XSLT Working Groups. *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. 2010. https://www.w3.org/TR/xquery-operators/ .

[23] W3C XQuery and XSLT Working Groups. *XSLT 2.0 and XQuery 1.0 Serialization (Second Edition)*. 2010. https://www.w3.org/T R/2010/REC-xslt-xquery-serialization-20101214[5] .

[24] W3C XQuery and XSLT Working Groups. *XQuery and XPath Full Text 1.0*. 2011. https://www.w3.org/TR/xpath-full-text-10/ .

[25] W3C XQuery and XSLT Working Groups. *XQuery and XPath Full Text 1.0 Test Suite*. 2010. https://dev.w3.org/2007/xpath-fu ll-text-10-test-suite[6] .

[26] W3C XQuery and XSLT Working Groups. *XML Path Language (XPath) 3.0*. 2014. https://www.w3.org/TR/xpath-30/ .

[27] W3C XQuery and XSLT Working Groups. *XQuery and XPath Data Model 3.0*. 2014. https://www.w3.org/TR/xpath-datamodel-30/ .

[28] W3C XQuery and XSLT Working Groups. *XPath and XQuery Functions and Operators 3.0*. 2014. https://www.w3.org/TR/xpath-functions-30/ .

[29] W3C XQuery and XSLT Working Groups. *XSLT and XQuery Serialization 3.0*. 2014. https://www.w3.org/TR/xslt-xquery- serialization-30/[7] .

[30] W3C XQuery and XSLT Working Groups. *XQuery and XPath Full Text 3.0*. 2015. https://www.w3.org/TR/xpath-full-text-30/ .

[31] W3C XQuery and XSLT Working Groups. *XML Path Language (XPath) 3.1*. 2017. https://www.w3.org/TR/xpath-31/ .

[32] W3C XQuery and XSLT Working Groups. *XQuery and XPath Data Model 3.1*. 2017. https://www.w3.org/TR/xpath-datamodel-31/ .

[33] W3C XQuery and XSLT Working Groups. *XPath and XQuery Functions and Operators 3.1*. 2017. https://www.w3.org/TR/xpath-functions-31/ .

[34] W3C XQuery and XSLT Working Groups. *XSLT and XQuery Serialization 3.1*. 2017. https://www.w3.org/TR/xslt-xquery- serialization-31/[8] .

[5] https://www.w3.org/TR/2010/REC-xslt-xquery-serialization-20101214
[6] https://dev.w3.org/2007/xpath-full-text-10-test-suite
[7] https://www.w3.org/TR/xslt-xquery-serialization-30/

[35] XQuery and XSLT Extensions Community Group. *XQuery and XSLT Extensions Community Group*. 2018. https://qt4cg.org/ .

[36] XQuery and XSLT Extensions Community Group. *qt4tests*. 2018. https://github.com/qt4cg/qt4tests .

[37] XQuery and XSLT Extensions Community Group. *XSLT 4.0 Test Suite*. 2018. https://github.com/qt4cg/xslt40-test .

[38] IETF/W3C XML Signature Working Group. *Canonical XML Version 1.0*. 2001. https://www.w3.org/TR/2001/REC-x ml-c14n-20010315.html[9] .

[39] Java Community Process. *JSR 225: XQuery API For Java (XQJ)*. 2009. https://jcp.org/en/jsr/detail?id=225 .

[40] Java Community Process. *JSR 63: Java API for XML Procesing 1.1*. 2001. https://jcp.org/en/jsr/detail?id=63 .

[41] BaseX. *BaseX GitHub - W3C Tests*. 2026. https://github.com/BaseXdb/basex/tree/main/basex-tests/src/main/java/org/basex/tests/w3c[10] .

[42] Leo Woerteler. *FITS BaseX*. 2013. https://github.com/LeoWoerteler/fots-basex .

[43] Evolved Binary. *Elemental GitHub - xqts Module*. 2026. https://git hub.com/evolvedbinary/elemental/tree/elemental-7.6.0/exist-xqts[11] .

[44] Evolved Binary. *X Test Harness (XTH)*. 2025. https://github.com/evolvedbinary/xth .

[45] Adam Retter. *W3C XQTS driver for eXist-db*. 2022. https://github.com/eXist-db/exist-xqts-r unner[12] .

[46] The eXist-db Authors. *eXist-db GitHub - XQTS to Junit*. 2019. https://github.com/eXist-db/exist/tree/develop-4.x.x/exist-core/src/test/java/org/exist/xquery/ xqts[13] .

[47] The eXist-db Authors. *eXist-db GitHub - W3CTests*. 2012. https: //github.com/eXist-db/exist/tree/eXist-2.0.x/test/src/org/exist/w3c/tests[14] .

[48] ETH Zurich. *W3C test suite implementation for RumbleDB*. 2026. https://github.com/RumbleDB/rumble-test- suite[15] .

[8] https://www.w3.org/TR/xslt-xquery-serialization-31/
[9] https://www.w3.org/TR/2001/REC-xml-c14n-20010315.html
[10] https://github.com/BaseXdb/basex/tree/main/basex-tests/src/main/java/org/basex/tests/w3 c
[11] https://github.com/evolvedbinary/elemental/tree/elemental-7.6.0/exist-xqts
[12] https://github.com/eXist-db/exist-xqts-runner
[13] https://github.com/eXist-db/exist/tree/develop-4.x.x/exist-core/src/test/java/org/exist /xquery/xqts
[14] https://github.com/eXist-db/exist/tree/eXist-2.0.x/test/src/org/exist/w3c/tests
[15] https://github.com/RumbleDB/rumble-test-suite

[49] Saxonica. *qt4tests - Saxon EE runner*. 2018. https://github.com/qt4cg /xslt40-test/tree/master/runner/saxon[16] .

[50] Saxonica. *Saxon FO Test Suite Driver*. 2011. htt ps://github.com/qt4cg/qt4tests/blob/master/drivers/saxon/FOTestSuiteDriver.java[17] .

[51] zorba.io. *Zorba FOTS Test Driver*. 2016. https://git hub.com/zorba-processor/zorba/blob/master/ctest2junit.cmake#L35[18] .

[52] Eric S. Raymond. *The Cathedral and the Bazaar*. 1999.

[53] Saxonica. *Saxonica - Our Products*. 2026. https://www.saxonica.com/products/produ cts.xml[19] .

[54] Open Test Alliance for the JVM. *Open Test Reporting - Event-based format*. 2026. https://github.co m/ota4j-team/open-test-reporting#event-based-format[20] .

[16] https://github.com/qt4cg/xslt40-test/tree/master/runner/saxon
[17] https://github.com/qt4cg/qt4tests/blob/master/drivers/saxon/FOTestSuiteDriver.java
[18] https://github.com/zorba-processor/zorba/blob/master/ctest2junit.cmake#L35
[19] https://www.saxonica.com/products/products.xml
[20] https://github.com/ota4j-team/open-test-reporting#event-based-format

Ant Visualiser

Ari Nordström

Abstract

What do you do when trying to decode an Ant script from hell? Ten thousand lines of script linking all over the place? Ants and subants getting in the way? XML properties scattered everywhere? Well, if you are like me, you draw pictures. And if that isn't enough, you write some XSLT to help you do it.

1. Intro

Ant is an XML-based language, originally for building software but since used for all kinds of orchestration tasks from file manipulation to XSLT transforms, from running executables to running Java software.

1.1. Background

A recent client of mine was (and is) heavily into Ant. Their business is product data, and they use Ant, XSLT, and various tools to process XML exported from, and imported to, a relational product database. The exports are used to publish the content on the web, in PDF format, etc, whilst the imports are used to either upload new data or tweak data already in the database. Additionally there are various macros and other functionality to perform file operations, validate, generate thumbnails, etc. The volumes are quite large; several gigabytes of data is common. The data is updated often, requiring frequent imports and exports, and performance is important enough to warrant using streaming XSLT to speed things up.

Whilst surprisingly robust, the system has grown organically and the Ant scripts, in spite of being modular, are now huge. A single such script can easily contain 10,000 lines, and because similar processes exist for different customers with different PDM databases, the repository is more than 1 GB in size.

Imagine, then, being asked to fix bugs in these scripts. Where do you even start?

2. Getting to Know an Ant Script

Trying to understand the inner workings of an Ant script is hard work, even with well-organised, small scripts. Ant scripts are essentially a series of tasks known as targets. These can be called by each other or set as dependencies that must be run

first. They can be modularised, and it is possible to create macros that can then be run by the targets.

Ant properties are used to pinpoint file locations, executables, etc. They are immutable variables that, once set, cannot be changed throughout the run of a script. The properties can be set locally or in separate text files, but also in XML files (`xmlproperty`), offering a more semantic approach to property hierarchies through element names and hierarchies. For example:

```
<catalogs>
    <dir location="${repos.foo}/catalogs"/>
    <catalog location="${catalogs.dir}/catalog.xml"/>
</catalogs>
```

This XML property fragment defines the location of a `catalogs` folder and the catalog file, `catalog.xml`, inside the folder. The folder location is set in `catalogs/dir/@location="${repos.foo}/catalogs"`, that is, the location consists of a property `${repos.foo}` defined elsewhere, appended with a subfolder `/catalogs`. The XML hierarchy itself declares an additional property for the folder location, however, namely `${catalogs.dir}` (because the "path" to `@location="${repos.foo}/catalogs"` implied by the XML property file is `catalogs.dir`).

We also define the location of the catalog file itself in a similar manner, in `${catalogs.catalog}`.

This is incredibly useful and very helpful when reading an Ant script, since a well-constructed XML property file can help decode what might otherwise be an obscure property name.

Ant allows us to use any number of such property files, and declare a property value any number of times, but only the first occurrence of a property is registered. Ant properties are immutable, so once set, later occurrences are ignored. This can be used to great advantage when, for example, defining a local set of properties that override some defaults:

```
<!-- Build properties -->
<xmlproperty
    file="build-db2html.properties.local.xml"
    keeproot="false"
    collapseattributes="true"
    semanticattributes="true"/>
<xmlproperty
    file="build-db2html.properties.xml"
    keeproot="false"
    collapseattributes="true"
    semanticattributes="true"/>
```

Here, if both `xmlproperty` files contain the same property names, the first encountered property is used.

There are many, many Ant extensions, for example, for various XML-based tasks, but perhaps the most important one is known as "Ant-Contrib". This is a JAR file that extends the language itself.

3. Introducing Ant Visualiser

Imagine, then, to be presented with multiple scripts, each of them thousands of lines and each linking to other modules, each setting build properties in places that are anything but obvious. How do you get to know them?

I like trees, and I like visual representations of trees. XML documents, of course, are trees by definition, so presenting them as tree diagrams is very helpful. For example:

```
book
└── chapter
    ├── title
    ├── para
    └── list
```

This, to me, is intuitive and obvious. If you're not a visual person, this says that a book contains a chapter that contains a title followed by a para and a list. (By the way, this is generated using the `tree` command in a Bash shell, and what we actually see here is a directory hierarchy.)

Wouldn't it be great if those huge Ant files could be presented as tree diagrams, excluding most of the code but making sure to keep links to other files, property definitions, targets, target dependencies, etc? A text-based tree diagram is one option, of course, but there are better alternatives.

I'm a big fan of mind mapping software. Freeplane[1], for example, is open source and my go-to solution whenever I need to structure something. A book XML structure in Freeplane might look like this:

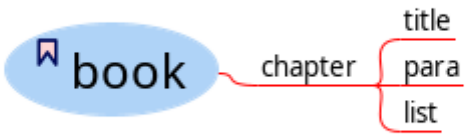


Figure 1. A Simple Book Hierarchy

[1] https://sourceforge.net/projects/freeplane/files/

The mind map file format is XML, so the above looks like this:

```
<map version="freeplane 1.12.1">
    <node TEXT="book" FOLDED="false" ID="ID_696401721"
CREATED="1610381621824" MODIFIED="1774896129945" STYLE="oval">
        <!-- Formatting removed -->
        <node TEXT="chapter" POSITION="bottom_or_right"
ID="ID_213855065" CREATED="1774896091588" MODIFIED="1774896127201">
            <edge COLOR="#ff0000"/>
            <node TEXT="title" ID="ID_888847201" CREATED="1774896114533"
MODIFIED="1774896116823"/>
            <node TEXT="para" ID="ID_740119509" CREATED="1774896117144"
MODIFIED="1774896119219"/>
            <node TEXT="list" ID="ID_901622521" CREATED="1774896119584"
MODIFIED="1774896121358"/>
        </node>
    </node>
</map>
```

There's much more markup than this -- formatting information, special features, etc -- but the core format is about as simple as it gets.

Can we convert the relevant parts of the Ant script to this format while getting rid of anything that isn't useful for a visualisation? It seemed to me that an XSLT would be enough.

4. XSLT

Much of an Ant script is actually *doing* something rather than declaring properties or linking to modules. My first draft therefore got rid of all that -- be it a little something applying a regular expression to a file, running an executable, looping over values, or something else -- but I've changed my mind since and will actually keep some of that information; it is helpful to know what a target does, after all. My current thinking is to summarise the functionality, somehow, and hide that summary by default.

My first draft focussed on visualising targets and their dependencies. For example, this target runs the targets listed in `@dependencies`:

```
<target
    name="all"
    description="Run an end to end SGML to XML conversion"
    depends="mkdirs, prepare, uncompress, preprocess-content,
    loop-spam, loop-sx, add-entity-declarations, loop-unparsed,
    run-manifest, copy-graphics, housekeeping"/>
```

This one is an Ant script that converts SGML files to XML. Several of the targets call other targets. For example, `loop-sx` loops its SGML input files, calling

another target (`target="sx"`) that does the actual work (converting to XML syntax, incidentally):

```
<target
    name="loop-sx"
    description="Loop through normalised files">
    <foreach
        target="sx"
        param="file"
        inheritall="true">
        <path>
            <fileset dir="${base.spam}" casesensitive="false">
                <include name="**/*.sgm"/>
                <include name="**/*.sgml"/>
            </fileset>
        </path>
    </foreach>
</target>
```

This, of course, means that the `loop-sx` node in the mind map should have `sx` as child. Etc.

All in all, visualising targets is easy enough and we get something like this:

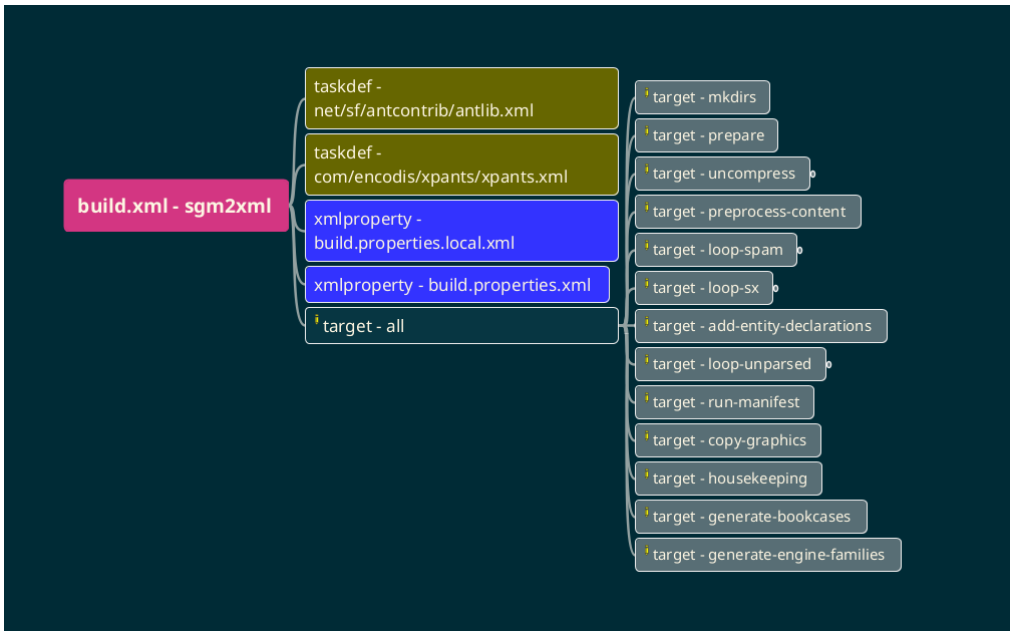


Figure 2. Build as Mind Map Test

This was just warmup, though, because I then looked at representing Ant calls and module inclusions. For example, we might have something like this:

```
<include file="ant/xproc.xml"/>
```

This is easy, of course; we can represent the `xproc.xml` module as a child node. All we have to do is look into that module with `doc(ant/xproc.xml)` and apply templates in whatever it contains. But Ant scripts very frequently include things using properties:

```
<include file="${antlibs.xproc}"/>
```

The XML property needs to be resolved, of course. We might find a match in an XML property file:

```
<antlibs>
    <xproc location="${libs.ant}/xproc.xml"/>
</antlibs>
```

This isn't as helpful as we might hope. The location consists of *another* property, appended with the Ant XProc module filename, so off we go again. Elsewhere in the XML property file, we find this:

```
<libs>
    <dir location="/my/libs"/>
    <ant location="${libs.dir}/ant"/>
</libs>
```

This should allow us to fully resolve the location of the XProc module, i.e. `/my/libs/ant/xproc.xml`. We should be able to open that XML file... or? Actually, no. Remember how Ant allows us to declare those variables any number of times, but since they are immutable, we need to find the first and stick to that?

For each Ant file we examine that links to something else, the links could well require parsing the properties available at that point; the links may use properties that need to be resolved before the link can be traversed. Etc. Consider, for example, the following example Ant build with four `xmlproperty` links:

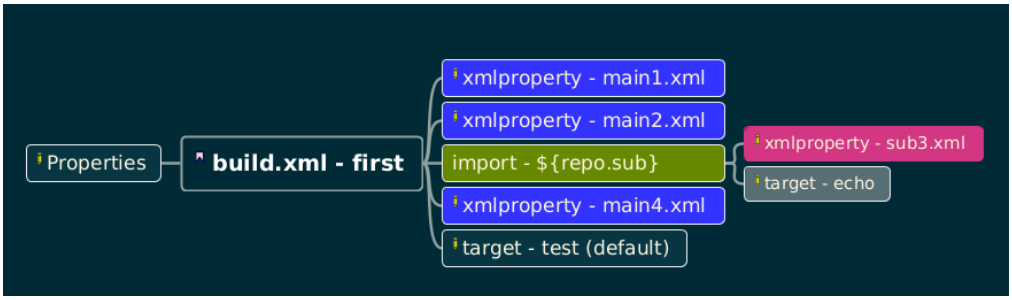


Figure 3. Ant Test Build with Properties

The `xmlproperty` files define various properties, some of them multiple times and with differing values. We need to resolve them in sequence, first to resolve the `import` link to a sub build and then to resolve any `xmlproperty` properties defined in it.

The approach is as follows:

1. Traverse through the current Ant build file to find any and all `xmlproperty` references to XML property files.

   Also, log any non-XML properties at the same time; both `property` elements in Ant *and* references to text-based link property files[2].

2. Open the `xmlproperty` and text-based property files in document order. Match every element with a `@location` or `@value` attribute and, for each, parse the ancestors to construct the property name. For example, matching `third/location="foo"` in the following structure, we get the Ant property `${first.second.third} = "foo"`:

   ```
   <first>
       <second>
           <third location="foo"/>
       </second>
   </first>
   ```

3. There's no need to keep the hierarchy; each property is flattened:

   ```
   <property path="first.second.third" value="foo"/>
   ```

   As the property may have been defined again in the XML property files that follow, the resulting list of properties could well have duplicates.

4. Having read the first XML property file in the current Ant script, repeat the process with the next one until done with the current Ant script.

5. The resulting list of properties may have duplicates, but also property values that combine literals with properties:

   ```
   <property path="fourth.fifth" value="${first.second.third}/bar"/>
   ```

   We therefore recursively parse the list of properties, resolving each property in a value, until done. A helper attribute `@done` is added to each `property`: `@done="true"` means that the value is a literal, and `@done="false"` means that the value still has one or more properties left in the value:

   ```
   <property path="fourth.fifth" value="${first.second.third}/bar"
   done="false"/>
   ```

   If the list has one or more property values with `@done="false"`, we need another iteration.

   The image below shows how unresolved `xmlproperty` properties are represented in the resulting mind map for a specific node. The table is a popup, appearing only when hovering a mouse pointer over the node. And yes, we also include the fully resolved properties. Read on.

[2]Not supported at the moment.

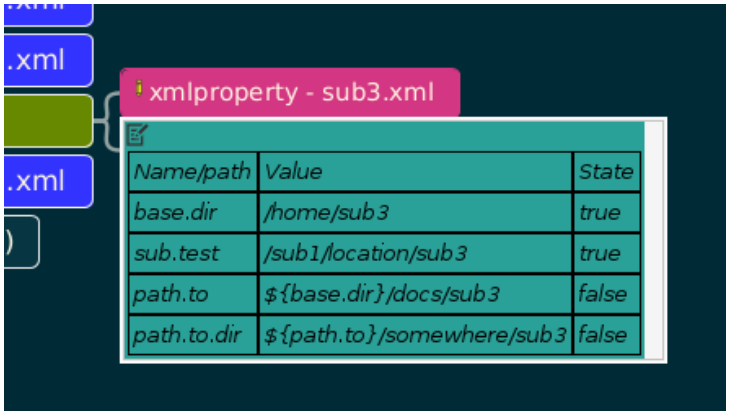


Figure 4. Unresolved Properties Marked with @done

6. Once the list of properties resulting from the current Ant script is normalised, we move on to look for any `import` statements, resolving any properties in the import reference, and then open the target and start the process of normalising any XML properties in the import target, providing the current sequence of normalised properties as input.

   The above process is repeated and, once done, we add any normalised new properties to the property sequence and provide it as input to the next imported Ant script.

   Rinse and repeat.

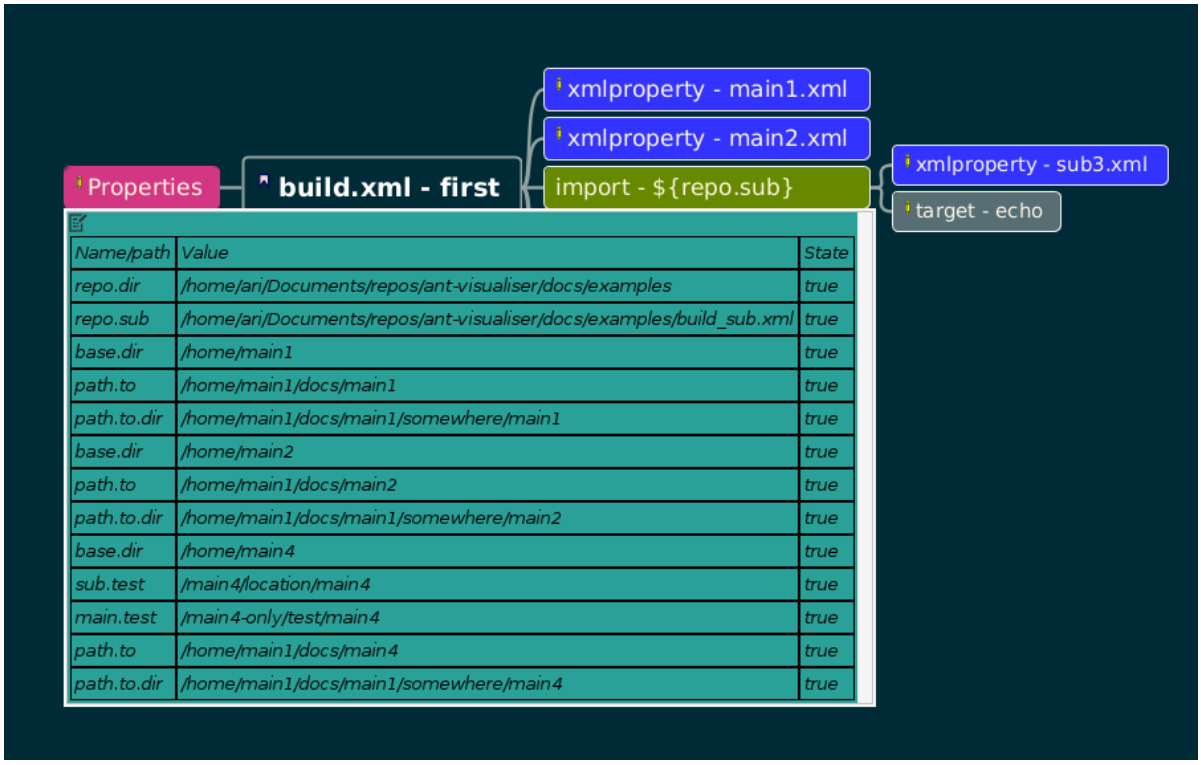
The resulting, fully resolved, list of properties is presented as a *Properties* node placed on the left-hand side of the map root. Again, it's available as a popup, see Figure 5.

Currently, the `xmlproperty` handling seems to work well.

5. In Closing

There are a number of considerations, improvements, and changes I would like to see:

- Representation for anything that doesn't follow a link that needs traversing. Think the everyday instructions that we fill our Ant builds with.
- Text-based property files are not supported. A target property file could be wholly text-based, and I don't (yet) address those at all.
- A pipelined approach using XProc could prove helpful, because the current XSLT does everything on the source tree. There's a lot of recursion and modes, and a pipelined approach would allow me to do things step by step, for example, to annotate the build to add all kinds of useful information.
- Speaking of pipelines, a script such as Ant Visualiser could be used to visualise an XProc pipeline. I thought about this, briefly, and it's a can of worms.



Name/path	Value	State
repo.dir	/home/ari/Documents/repos/ant-visualiser/docs/examples	true
repo.sub	/home/ari/Documents/repos/ant-visualiser/docs/examples/build_sub.xml	true
base.dir	/home/main1	true
path.to	/home/main1/docs/main1	true
path.to.dir	/home/main1/docs/main1/somewhere/main1	true
base.dir	/home/main2	true
path.to	/home/main1/docs/main2	true
path.to.dir	/home/main1/docs/main1/somewhere/main2	true
base.dir	/home/main4	true
sub.test	/main4/location/main4	true
main.test	/main4-only/test/main4	true
path.to	/home/main1/docs/main4	true
path.to.dir	/home/main1/docs/main1/somewhere/main4	true

Figure 5. Resolved Properties

- There's a part of me that wants to implement a pipeline for the Ant Visualiser using Ant. How meta is that?
- It also occurs to me that an Ant script could be represented as a knowledge graph, using those same scripts to resolve properties, but the end result would be a very different animal.

  My goal with the Ant Visualiser was simply to get an overview of a big Ant script, understand how it was structured, and how it referenced other scripts and properties. A graph approach would make the visualisation more dynamic and more about context, rather than a tree branching off in imports, targets, etc.

Maybe version 2.0.

Bibliography

[1] “Ant Visualiser” [online, fetched on 26 May 2026] https://github.com/sgmlguru/ant-visualiser

Jewels in Plain Sight: Building CREPDL Tooling with AI assistance

Eamonn Neylon
SignalArc
<eamonn@signalarc.com>

Abstract

Use of a Large Language Model to build a tool for character use validation is presented. This neglected application area and the process by which the tool was developed is described. An accompanying library of character repertoire schemas to test the tool, and a Schematron-based quality tool were also developed. Building on the iterative approach used, consideration is given to the value of precise specifications for code generation. The approach can be used for rapid prototype tooling to surface defects in standards under development, and to change the feasibility of developing supporting tooling to make standards more useful to their prospective users.

1. Introduction

Organisations process text for a range of purposes, such as in data entry pipelines, archival systems, typesetting workflows, and data interchange. These applications routinely need to assert that a body of text uses only characters appropriate to a given language, script, or context. The Character REPertoire Description Language (CREPDL)[1] provides a formal statement language to express and check such assertions. Its problem domain is widespread.

Also known as ISO/IEC 19757-7:2020, CREPDL is a standard for declaring, exactly which Unicode[2] specified code points are permitted in textual content. Published in 2007, the CREPDL standard was revised in 2020. It belongs to the Document Schema Definition Language (DSDL) family, a suite of complementary standards for validating structured documents and content. The reference implementation for CREPDL is Murata Makoto's open-source processor which can be compiled using Microsoft Visual Studio, available from the CITPCSHARE GitHub repository[3].

Yet for nearly two decades, CREPDL had almost no public tooling beyond Murata's reference implementation. The reason was not a flaw in the standard. The reason is a combination of awareness, tooling availability and integration options that made it easier to ignore the problem or to use other tools to achieve similar outcomes. This paper introduces a user-accessible, web-deployable implementation — with a schema library, validation UI, and developer-friendly

interface that has been developed using generative language model assistance. Such an approach allows development that was previously too costly to justify without an obvious commercial return. The cost equation has changed with the widespread availability of agentic AI to assist with implementation.

The work described herein is available as a public service called CREPDL·CHECK[4]. It is freely available for public and commercial use by anyone interested. The source code for CREPDL-CHECK is available on GitHub[5]. A complimentary service has also been developed to allow fonts to be analysed for their coverage of character sets including those defined by CREPDL schemas.

The large language model (LLM) used to develop the tools discussed is Anthropic's Claude[6]. This paper does not consider the relative merits or otherwise of using any particular model. Rather it addresses the process of using such a tool and reflections on the strengths and weaknesses of the overall approach. The use of this approach for assisting with specification-driven implementation tasks is considered. Other LLMs appear to be capable of similar outcomes.

2. Understanding Character Repertoires

Character Repertoires are an expression of what text characters can exist in a particular context. Character repertoires do not presuppose that the character will exist in an XML document but instead describe what is expected in any sequence of character encoded as text. In the context of the CREPDL specification, the character repertoire uses set operations over groups of characters (represented as a XML document). Although this sounds complicated, only three operations exist, which means that CREPDL is easy to learn.

Before discussing the operations, let us be clear what a character is. There are many ways to represent human languages in computer systems. When there is no consensus on how characters are to be encoded, it becomes difficult to exchange text with other parties. Character encoding protocols are necessary to know that for instance the hexadecimal representation of an uppercase Roman 'A' is 41 (this is true both for ASCII encoding and Unicode). Once the encoding is known then rendering the character becomes possible, so that humans can recognise it. The character is the encoding in a particular encoding scheme. It is not the rendering of that character.

CREPDL relies upon the Unicode standard for encoding characters. Unicode acts as a registry describing how to encode characters from all the world's languages. Unicode normalises the use of character so that languages can coexist in the same encoding without clashes of usage/assignment (known as code points). Managing such assignment of code points and ranges is a massive endeavour. The Unicode Consortium exists to ensure appropriate and equitable assignment of code points.

CREPDL builds on the work of the Unicode Consortium, which brings a burden of maintenance to implementations. This is because the allowed encodings for Unicode are updated with each new release of the Unicode specification regularly. So, it is important to be clear which version of Unicode is intended within a CREPDL description of allowed characters. This leads to attributes in the schema which are important to the interpretation of the specified repertoire.

The following shows a minimal, best-practice-conformant CREPDL schema:

```
<crepdl:collection xmlns:crepdl="http://purl.oclc.org/dsdl/crepdl/ns/
structure/2.0"
                   xml:lang="en"
                   minUcsVersion="15.1"
                   maxUcsVersion="16.0">
                  <crepdl:kernel>[\x{0041}-\x{005A}\x{0061}-\x{007A}]</
crepdl:kernel>
                </crepdl:collection>
```

Recognising the encoding of a character A it is possible to guess that this CREPDL schema is for the English language characters (there is also a clue in the use of the xml:lang attribute although it is debatable whether this is a legitimate use since there is no natural language within the schema. it should be apparent that writing CREPDL schemas requires familiarity with the Unicode specification as its principal area of expertise. CREPDL is unlikely to be known amongst its prospective communities of interest.

CREPDL's character class expressions are Unicode regular expressions. As mentioned earlier, the evaluation of whether a character is a member (included in a range) depends on which version of the Unicode Character Database is in use. Without explicit version specification, two conformant processors may disagree on the membership of a character that was added in an intermediate Unicode version. Therefore, the root element should carry a `@minUcsVersion` attribute (e.g. `minUcsVersion="15.1"`). There is also an optional `@maxUcsVersion`. Where both are present, `minUcsVersion` should not be greater than `maxUcsVersion`, otherwise there is an authoring error.

RELAX NG is used to specify a schema for CREPDL specifications. The CREPDL specification is expressed as an XML document that conforms to the CREPDL standard. The following is the governing schema as provided in the standard:

```
default namespace = "http://purl.oclc.org/dsdl/crepdl/ns/structure/2.0"
start = coll
coll = union | intersection | difference | ref | repertoire | char
union = element union { commonAtts, coll+ }
intersection = element intersection { commonAtts, coll+ }
difference = element difference { commonAtts, coll+ }
ref = element ref { commonAtts, attribute href { xsd:anyURI} }
```

```
repertoire = element repertoire { commonAtts, attribute registry
{ text },
             attribute version { text }?, (attribute name {text } |
attribute number { xsd:int} ) }
char = element char { commonAtts, (text |
       element kernel { commonAtts, text} |
       element hull { commonAtts, text} | (element kernel { commonAtts,
text},
       element hull { commonAtts, text} ) ) }
commonAtts = attribute minUcsVersion { text }?,
             attribute maxUcsVersion { text }?,
             attribute mode { "character" | "graphemeCluster" }?
```

This is the entire text of the schema from the standard. Within the standard, the accompanying license states:

> *Permission is hereby granted, free of charge in perpetuity, to any person obtaining a copy of the Schema, to use, copy, modify, merge and distribute free of charge, copies of the Schema for the purposes of developing, implementing, installing and using software based on the Schema, and to permit persons to whom the Schema is furnished to do so, subject to the following conditions:*
>
> *THE SCHEMA IS PROVIDED 'AS IS WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OR CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SCHEMA OR THE USE OR OTHER DEALINGS IN THE SCHEMA.*

Due to the cost of the standard, awareness of this license will be low. Indeed, the author did not realise such use was possible when starting to build the tool described herein.

The CREPDL standard set operations are union, intersection, and difference. The union operation includes all character in all ranges covered. The intersection operation includes only shared character in all range covered. The difference operation excludes shared character in the ranges covered. This means that sets of character can be combined to various effects - for instance a punctuation set can be combined with the characters used in a language, two languages can be combined to check language dictionaries, or characters that overlap in two sets can be excluded, for example, to remove punctuation before indexing.

CREPDL logic is not Boolean; rather a three-valued membership model is used. A character that falls within the kernel expression is definitively in the repertoire; a character within the hull but outside the kernel is of unknown status

(it may be acceptable, but the schema does not assert so); a character outside the hull is definitively not in the repertoire. This is a meaningful semantic refinement for languages with disputed or evolving orthographies, but it requires careful implementation: the three-valued logic must propagate correctly through union, intersection, and difference operations.

3. Building the CREPDL Validator

3.1. Technology Choices

The development of the tool commenced on 9 March 2026 with two prompts used to with Anthropic's Claude Sonnet 4.6 Large Language Model (LLM). The prompts were:

> *Create an ISO/IEC 19757-7:2020 (CREPDL) schema for the Irish language that would detect any non-Irish characters in a text*
>
> *create a tool that allows checking of a text instance against a crepdl schema*

The prompts are generic except for the specification of the standard from which the CREPDL schema for testing. In hindsight, it would have been better to supply the RELAX NG schema from the specification as despite the specific reference Claude struggled to produce a correct form of the schema (although this was not immediately obvious until testing was undertaken).

For the tool, the LLM initially developed a browser-based solution using JavaScript. While this solution appeared to behave as expected, Claude was directed to instead create a Java servlet for deployment. This change was to make the code easier for the author to evaluate. However, when testing started another reason for using Java emerged: several mistakes were captured because of Java's strong typing system which might not have been picked up without the compilation stage that a Java-based approach necessitated. Other reasons to prefer Java are the ecosystem for XML processing being mature, standardised, and well-documented, and the native Unicode handling in Java which is vital for correct handling of character properties that CREPDL validation requires.

Building of the servlet used the maven build tool and the resulting servlet deployed into a Tomcat 10 environment locally for testing. Although it is possible to ask Claude to perform these operations, running through these steps manually helped the author spot issues in the generated artefacts. Rather than let the LLM correct its own mistakes without notification, it is illuminating to see where it hallucinates information or missed essential parts of the standard. The imprecision of the LLM meant that many iterations were necessary to get the tool functioning as expected.

3.2. Iterative Progress

The Java output was consistently idiomatic. However, there were mistakes reading the specifications. The LLM seemed to be mixing up requirements from different versions of the standard. The agentic 'hallucinations' became apparent, with the LLM constructing its own namespace for the CREPDL schema and having trouble producing testable CEPDL schemas.

The early iterations of the validator were faulty. There were issues interpreting what was in the specification. It was only when checking the initially generated CREPDL schemas that the extent of the mistake was noticed. This would have been improved if the standard and included schemas were supplied as part of the initial prompt. Supplying the RELAX NG[7] schema would have been permitted under the terms of its license, but the author was not aware of this at the time.

Addressing the issues required familiarity with the specifications and interactive research during the development. Scepticism is useful when using the tools to protect against assumptions of correctness by the LLM. Instead providing specific feedback allows the fixing of issues as they arise. Despite the issues, after many iterations, a useful tool emerged. The LLM had produced a tool that could parse a CREPDL XML schema document, constructing the in-memory character repertoire from char, range, union, intersection, and difference elements; iterating over input text at the code-point level; and emitting the three-valued validation result for each code point.

Validation results are presented to support user understanding, not merely pass/fail reporting. A character violation does not emit a raw code-point failure; it produces an annotated preview identifying which characters failed and providing enough context to understand why. The three-output tab structure (Violations, Annotated Preview, Statistics) will lead a new user from the most actionable information (what failed) to progressively richer analysis (where in the text, how many, what proportion of the total).

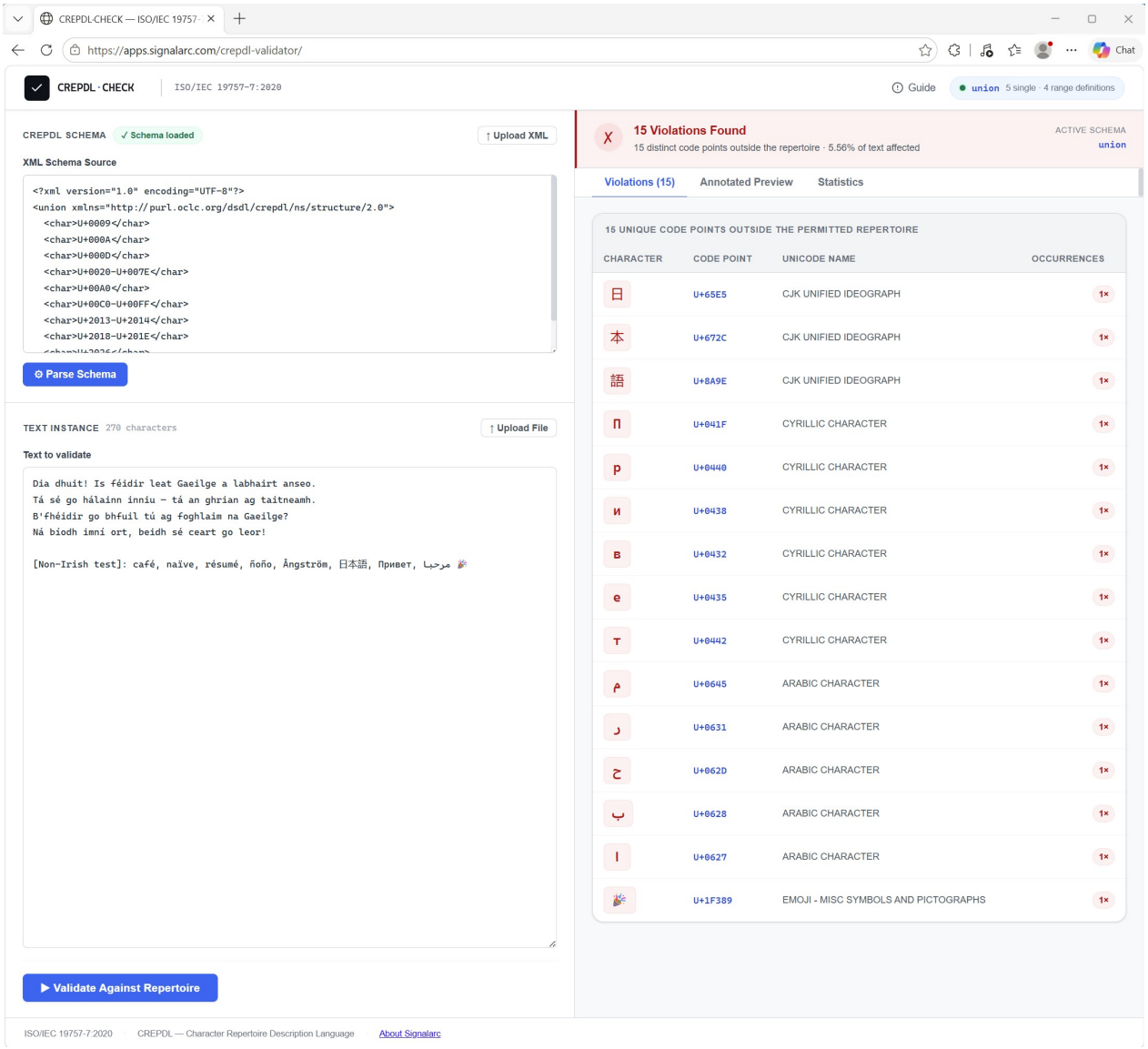


Figure 1. Figure 1: The CREPDL·CHECK application web interface

There are no restrictions as to the use that may be made of these resources. The author wishes to promote the use of CREPDL in all environments.

To compliment functional code generation, there are other facets of software development that agentic coding can support. It proved helpful to think of the LLM as a team with distinct capabilities that can be applied in layers to add refinement to a software project. This is often beyond the reach of real-world teams (for reasons such as scheduling, finance, or security).

3.3. Other Development Aspects

Once the application was functioning as expected, other aspects of the software process came to the fore. The first was to apply consistency to the branding of the solution. Early iterations used a black background which the author struggled

to read and was inconsistent with the deployment environment. Brand alignment was the first 'non-functional' area of solution iteration once the application was functional.

Accessibility is a primary requirement for the author. The tool went through more iterations that targeted WCAG 2.1 AA conformance. The initial poor accessibility of the generated solution is an inditement of common practices in website development which the LLM takes as normal before being directed to address the issues. The LLM is capable of greatly improving the accessibility, but the fact that it needs to be asked to do so is disappointing.

Concrete accessibility measures include: semantic HTML throughout the result regions; ARIA live regions for dynamic validation output so that screen reader users receive result updates without a page reload; keyboard navigability for all interactive controls; and sufficient colour contrast for violation annotations. Claude generated accessible markup when accessibility requirements were included explicitly in the prompt context — another illustration that upstream specification quality governs downstream output quality, whether the specification is an ISO standard or a WCAG[8] success criterion.

Another aspect that an LLM can assist with is security. Within CREPDL, the `ref` element allows a schema to include another schema by URI reference. The 2020 edition specifies that relative URI references are resolved against the base URI of the including document. In a web application context, where schema documents may be uploaded, stored temporarily, or referenced from a schema library, determining the correct base URI is non-trivial. This required explicit architectural decisions about how uploaded schemas are stored and how base URIs are assigned. Other security aspects remain to be investigated.

4. Testing Approaches

As part of this work, a set of generated CREPDL schemas has been made available as a schema library[9] which was also used to test the validator. Conformant CREPDL schemas were produced for over 80 languages systematically, drawing on Unicode block references and language-specific orthographic sources. The systematic nature of this work — schemas for each of 24 EU languages, for Cyrillic-script languages, for Arabic-script languages, and so forth — is where LLMs can be very productive.

For each schema, a description is associated that explains not just what each schema covers, but why particular diacritical characters or script blocks are included in the schema — information that makes the purpose of the schema concrete. This library should help users by providing an off-the-shelf set of starting points. Another set of schemas has been provided for testing in the CITPCSHARE GitHub repository.

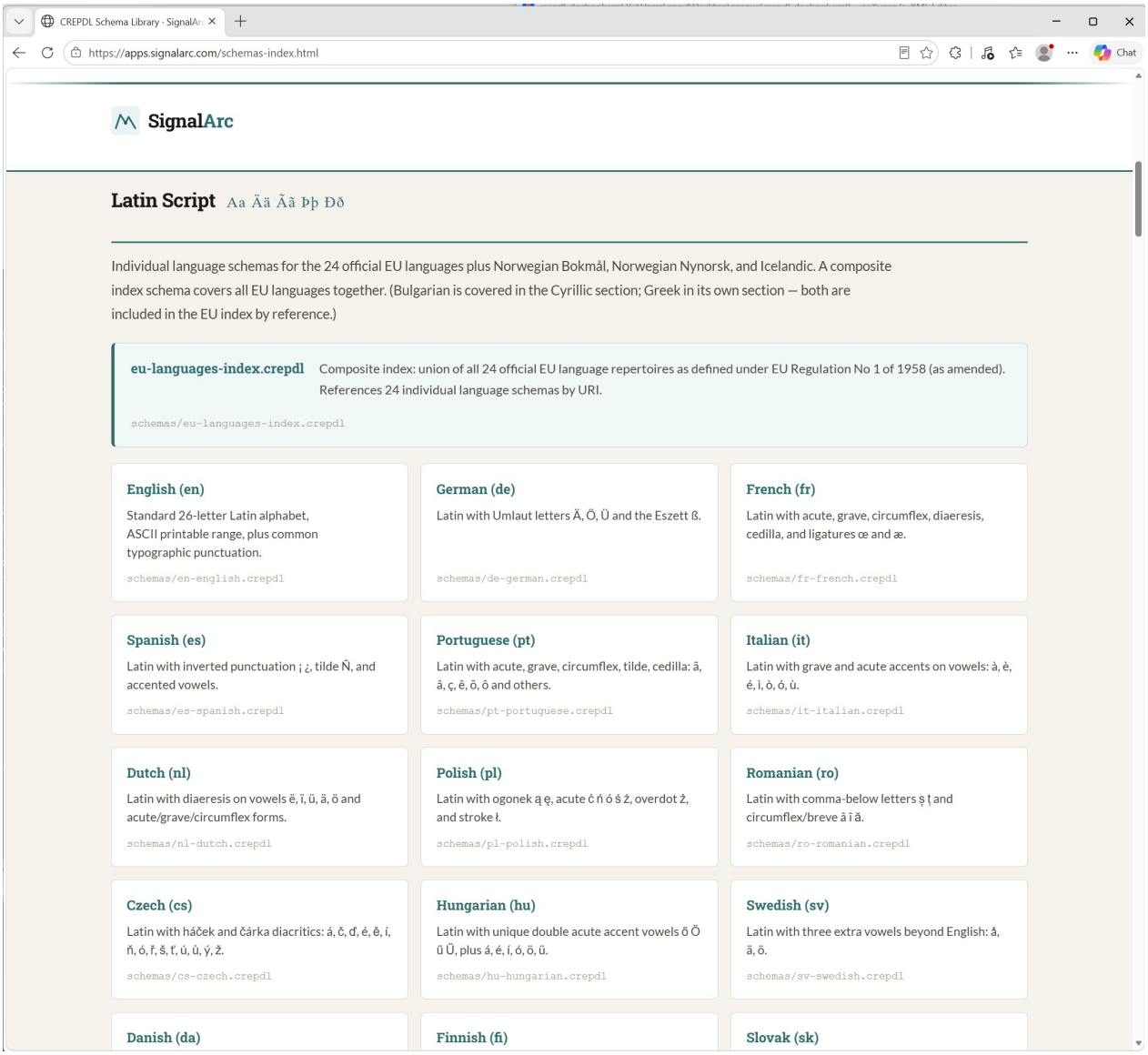


Figure 2. Figure 2 CREPDL schema library with over 80 schemas

As with all validation, there is a limit as to what to expect of the results. It is for users of a language to know what characters are used in practice and what punctuation is permitted in use. Such domain-specialization is common knowledge amongst speakers of a language or users of an application and so there will be a need for human verification of CREPDL schemas - at least during the early adoption of the technology.

Large language models are sometimes called 'stochastic parrots'. They use a probabilistic approach to the generation of human-like outputs through mimicry. This is seen when the models make up information (hallucinations) rather than interact with available resources (such as the prompter) to seek clarification. So, how to check the generated CREPDL schemas for 'correctness' or the absence of

errors? Fortunately for this work there exists mechanisms both automated and manual by which such affirmation can be achieved.

4.1. Schematron

The relationship between CREPDL and Schematron[10] is interesting. It is possible to achieve some of the validation that CREPDL supports using Schematron itself. Though less expressive (and nowhere near as elegant) such character range restrictions expressed in Schematron exist (likely due to lack of awareness and tooling to support the use of CREPDL). One example of such usage is the ISO Schematron for the Standard Tag Set which uses Schematron rules to express what could be achieved much more concisely in CREPDL.

For testing, the CREPDL tools it was vital that valid CREPDL schema instances and test data were available. The large language model could help with this (indeed generating tests is a problem that stochastic parrots are well suited to). But it is vital to be able to test the tests themselves. Schematron was used to ensure that the generated test CREPDL schemas were correct (in terms of how they are expressed, not that the right characters are used as that requires language skills).

Again the LLM was used to produce the tool for testing the validation outputs of the LLM. The resulting Schematron [11] checked the CREPDL schemas for several things, but by far the most significant is how the characters themselves are encoded. This closes a useful loop: CREPDL defines the character repertoires; the Schematron validates the CREPDL documents themselves.

Inside kernel, hull, and bare collection text nodes, non-ASCII characters should be written as XML numeric character references (一 rather than the literal character 一) to ensure the schema file is unambiguous in an editing environment and robust against encoding accidents. An XPath 2.0 regex matches against [^\x00-\x7F] to detect literal non-ASCII bytes in these contexts. This is particularly important for CJK and Arabic-script schemas, where literal character inclusion in the source file is a common but fragile practice.

The generated CREPDL Schematron[11] also warns when a collection element carries only a bare text node (no explicit kernel or hull children): this is legal but ambiguous, because the processor treats bare text as if both kernel and hull are identical. It also warns when only a hull is present without a kernel: in this case no character will ever be definitively accepted (every code point within the hull is "unknown"), which is rarely the intent. Finally, the Schematron warns when the kernel and hull expressions are textually identical, which is redundant — if the repertoire is exact, hull should be omitted. The Schematron also checks that the set operators have more than one child. Taken together, the patterns constitute a quality gate that a CREPDL schema should pass before being submitted to a processor.

4.2. Role of the Specification

In developing the validator, the text of the standard was not provided to the LLM as context. This was a deliberate choice because of the copyright restrictions on the standard (the author did not know of the embedded license for use of the RELAX-NG component at the outset). So the LLM used its training data and the fetch MCP to access knowledge of the CREPDL specification. This resulted in some issues where information from different versions of the specifications were used for different parts of the processing. These inconsistencies tended to show up as compile time issues, and so were to some extent autodetected through the semi-manual workflow used to develop the validator.

For those concerned about developing applications and the use of proprietary or confidential information for the training of online LLMs, an alternative does exist. Although requiring some technical investment over just accessing the online models, running models locally is a realistic alternative to supplying data to online services which seem to reposition their ethical compasses according to the expectations *du jour*. The use of local LLMs is recommended if sensitive content is supplied to a model, since control over how such information is propagated is brought back to the user who has invested in the knowledge of how to run such models locally.

Claude must have used information from both the CREPDL and Unicode specifications to develop the validator and the schema library. This is the essence of what is needed to produce a conformant implementation. The investment of expertise that went into writing the CREPDL standard is realized when the standard is used as input to construct a validator. The alternative of asking an AI to implement a validator based on a generic description or a poorly structured requirements document would clearly produce qualitatively worse results. A well-specified standard is no longer merely a reference document: it can also be viewed as executable content.

The most important lesson from this project is that specification quality directly determines LLM output quality. ISO/IEC 19757-7:2020 provided:

- a formal RELAX NG grammar for CREPDL schema document;
- explicit processing model semantics for each element type;
- a defined URI resolution model for the ref element; and
- a clear account of the three-valued membership model (in / out / unknown) introduced by the kernel/hull distinction in the 2020 edition.

So the specification itself both defines the expectations of the generated code and can be used to test that those expectations have been met.

5. Maintenance

By making the source code for the CREPDL CHECK validator available on GitHub, there is an implicit expectation that any user making a pull request against the git repository will have the proposed change considered. This changes the nature of what the code means to people who want to engage with it. A switch from automated generation to human curation becomes necessary to encourage people to engage with the project. That is not to say that there would be no use of LLMs on the validator in the future, but that future changes need to be accepted by the maintainer(s) of the project.

Such a switch-over in how the project is managed is considered necessary to encourage future human investment in the code and the accompanying resources. Since it has already been identified that human involvement is essential to check CREPDL schemas and that their existence and curation is fundamental to the confidence needed to use the validator effectively, it becomes apparent that the LLM's role is as an accelerator rather than a replacement that involvement. The type of expertise needed changes from knowledge about how to check conformity to knowing which conformities need to be checked, i.e., expertise around the use of language/characters rather than the technology. The domain verification work (checking repertoire accuracy) should be focused on correctness rather than syntax.

Making the validator available without the means of applying it would not be sufficiently encouraging for potential users. That is why the aforementioned schema library and an accompanying tutorial on CREPDL have also been provided. This teaching resource should help prospective users understand how they can make use of CREPDL for their own purposes. But how useful a validator is to someone who does not know they need a validator is debatable. To address this gap of knowledge another software tool was developed to create a bridge for potential users to the concepts of CREPDL.

For an underused standard like CREPDL, the first tool a user encounters is also their primary introduction to the standard itself. CREPDL·CHECK was designed with this explicitly in mind: the interface is an educational surface as much as a processing tool. But users do not think in terms of character encodings but it terms of content use. So the complimentary tool works with concepts that users are likely to have a better knowledge of - whether a font can be used to render their content.

The FontCheck tool[12] analyses an uploaded font to understand which glyphs are available to represent Unicode characters in a font. Users can check their text against the metrics captured about a font. Once a font is analysed, its characteristics are memorized so that other users who do not yet have the font can learn about its characteristics. The font check application is intended to encourage future users to become aware of the character set coverage of their

fonts and make them aware of the strengths and weaknesses of the fonts they are considering.

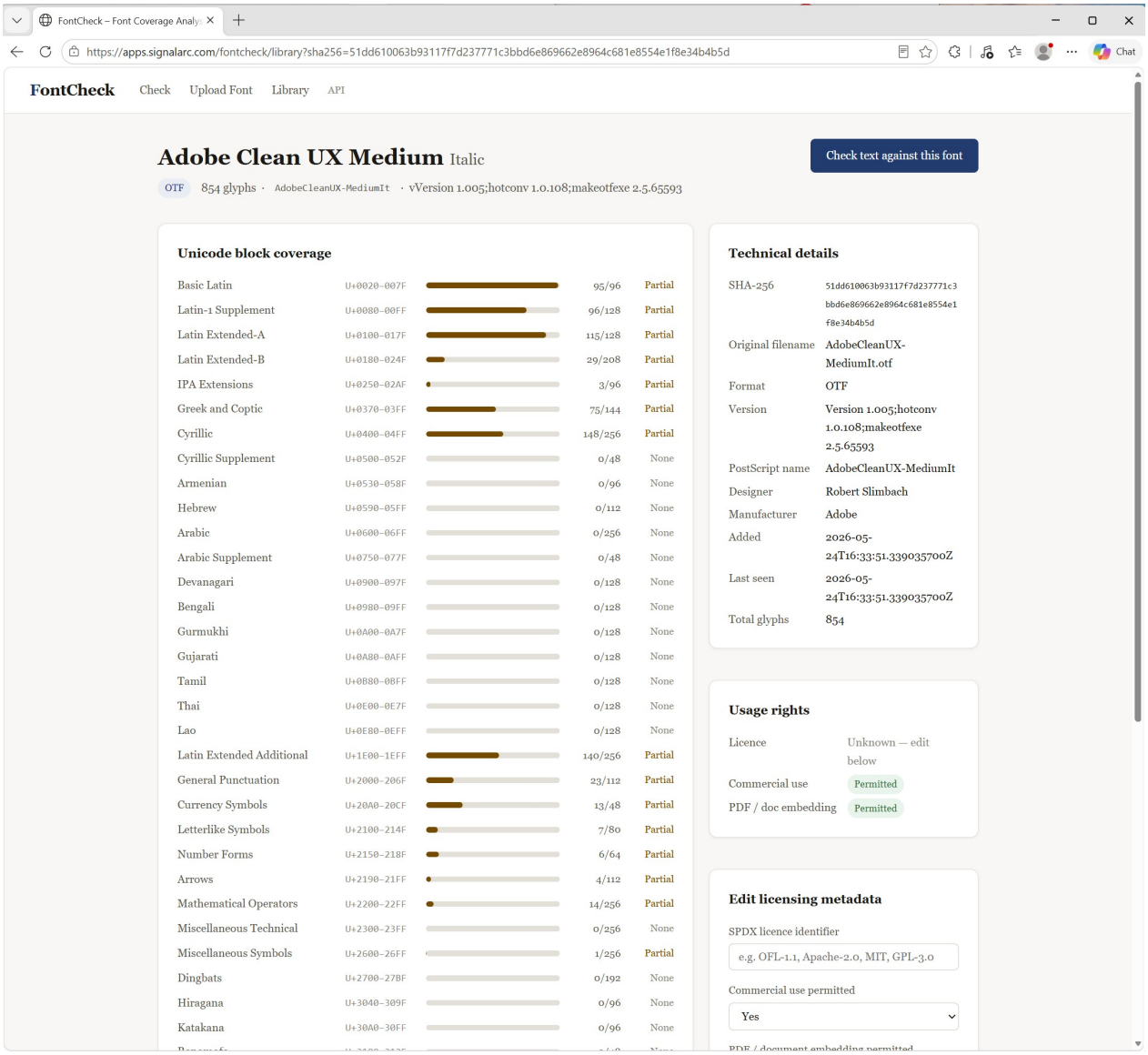


Figure 3. Figure 3: The font analysis view of the FontCheck application

6. Extrapolation

6.1. Niche Technical Work

CREPDL has existed for nearly two decades without an implementation accessible to non-specialist users. The standard is not wrong; the problem it solves causes real-world issues. The barrier to its use was economic in so much as the implementation cost made the work commercially unfeasible. But the emergence of LLMs changes the financial calculation by collapsing the cost of an initial

implementation. What remains is the irreducible cost of domain expertise — reading the standard carefully, verifying edge cases, auditing the output — and that cost is justified precisely because the problem is real.

The question shifts from whether an implementation is affordable to whether it is useful to do so. For CREPDL, the answer was always 'yes'. The same is likely to be true for many other standards that have languished without tooling to enable their adoption. The lack of citation of CREPDL from other standards illustrates how unfamiliar standards developers are with this standard - the only current standard making normative use of CREPDL is ISO/IEC 30114-2:2018 which is *Extensions of Office Open XML file formats Part 2: Character repertoire checking* from the same committee as the DSDL standards.

Table 1. Table 1. References to CREPDL in other international standards

Status	**Designator**	**Informatively References**	**Normatively References**
Definitive	PD ISO/IEC TR 24754-2:2011	ISO/IEC 19757-7	
Withdrawn	ISO/IEC 19757-3:2020	ISO/IEC 19757-7:2009	
Confirmed	ISO/IEC 30114-2:2018		ISO/IEC 19757-7

6.2. Candidates for Similar Work

The pattern identified of having a well-specified standard that has failed to reach its potential applies to a significant number of other standards. Some candidates for similar treatment from within the DSDL family of standards include:

- NVDL (ISO/IEC 19757-4): Namespace-based Validation Dispatching Language[13] which provides an elegant mechanism for applying different validators to different namespaces within a compound document (e.g. XHTML with embedded MathML and SVG). It has almost no public tooling despite being technically sound and addressing a genuine problem in multi-namespace XML authoring workflows.
- DSRL (ISO/IEC 19757-8): Document Schema Renaming Language[14] specifies default values for XML documents in a way that is schema-language-independent. A practical implementation could be integrated with existing XML editors or validators.

Beyond this familiar field there are likely to be many other standards that could be revived:

- ISO 8601-2:2019 — Extended date/time representations[15]: The extended profiles of ISO 8601 for uncertain and approximate dates (EDTF) have incomplete library support in many languages. A conformant validator and formatter would serve many communities.
- W3C EARL (Evaluation and Report Language)[16]: A formal RDF vocabulary for accessibility evaluation results with limited tooling for generating, consuming, or presenting EARL reports in accessible formats — an irony not lost on the accessibility community.
- ISO/IEC 14977 — EBNF[17]: The ISO standard for Extended Backus–Naur Form is rarely the basis for parser generators despite being the normative grammar notation for many other ISO standards. A conformant grammar tool that could directly process the EBNF appearing in ISO standards would be a useful bridge.

In each case, the methodology is the same: obtain the standard text; use it as structured input to an LLM; implement with human expert review; publish with documentation oriented toward both practitioners and newcomers to the standard. The schema library component — a curated, downloadable corpus of conformant schema instances — is a particularly transferable pattern: it lowers the barrier to adoption by giving users something to work with immediately, before they understand the schema language well enough to write their own.

6.3. Automated Testing of Standards

One benefit of building a conformant implementation with LLM assistance is the light it sheds on the standard being implemented. Implementing a specification requires a thorough reading of it. A LLM-assisted implementation, with its tendency toward literal interpretation, can surface ambiguities and under-specification that human readers miss.

CREPDL is a well-constructed specification. The process of asking the LLM to implement it and verifying the output against the standard would identify precisely where a standard is silent or ambiguous. This suggests a general methodology for standards developers:

- Prior to release, create an LLM-assisted implementation as a structured review exercise. This prototype need not be production-quality; its value lies in the questions it raises.
- Use any implementation questions as a structured checklist of specification completeness: every question that requires human judgment to answer is a candidate for a clarification in the standard.
- Test against worked examples from independent implementations to verify that the standard's semantics are deterministic — that two conformant processors will always agree on a given input.

- Iterate on draft standard text in the same way developers iterate on code: small changes, immediate verification, explicit regression testing.

The development of CREPDL·CHECK against the 2020 edition of the standard, cross-checked against [2], effectively constituted a conformance test suite construction exercise. Using this methodology before publication would produce a tighter, more implementation-ready specification. The cost of the exercise, which was previously prohibitive for niche standards, is now substantially reduced by LLM assistance.

7. Conclusions

Building CREPDL·CHECK and the accompanying schema library with LLM shows that a lot can be achieved with relatively modest resources. As anticipated before commencement, specification quality is the most important input. The precision of ISO/IEC 19757-7:2020 was the primary determinant of LLM output quality. A formal standard is better LLM context than a prose description; a prose description is better than nothing. As a corollary: Schematron is the right tool for schema quality enforcement. The companion Schematron, using ISO/IEC 19757-3 to validate ISO/IEC 19757-7 documents, is a practical and elegant application of the DSDL family's composability.

Tooling is necessary for education and therefore adoption. For underused standards, the first tool is the first introduction. Interface design, error messaging, and documentation are important to enable engagement. The human role shifts, it does not diminish. The work changes character; it does not go away.

LLM-assisted implementation can be a standards-testing methodology. Commissioning a prototype implementation before publication is a cost-effective method for identifying specification gaps and ambiguities. Niche is no longer a barrier. If the problem is real and the standard is good, the implementation is now achievable. The DSDL family deserves a second look.

It also changes the case for revisiting existing standards that were ahead of their time. Standards that were technically correct but unimplemented because of cost barriers deserve a second look. This work suggests a direct implication for standards development organisations: a well-written standard is now much closer to a working implementation. This changes the case for investing in specification quality, in worked examples, in normative test suites, and in reference implementations published alongside the standard rather than years later.

References

[1] *Document Schema Definition Languages (DSDL) — Part 7: Character Repertoire Description Language (CREPDL)*, 2nd edition. ISO/IEC, 2020. ISO/IEC 19757-7:2020

[2] Unicode Consortium *The Unicode Standard, Version 17*. Available at: https://www.unicode.org/versions/Unicode17.0.0/ [Accessed March 2026].

[3] Makoto Murata *CREPDL reference implementation (CITPCSHARE)*. Available at: https://github.com/CITPCSHARE/CREPDL [Accessed May 2026].

[4] *CREPDL·CHECK application*. Available at: https://apps.signalarc.com/crepdl-validator/.

[5] Source repository available at: https://github.com/signalarc/crepdl-validator

[6] Anthropic *Claude model documentation*. Available at: https://docs.anthropic.com

[7] *Document Schema Definition Languages (DSDL) — Part 2: Regular-grammar-based validation — RELAX NG*. ISO/IEC, 2008. ISO/IEC 19757-2:2008

[8] W3C *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C Recommendation, 2018. Available at: https://www.w3.org/TR/WCAG21/

[9] *CREPDL·Schema Library*. Available at: https://apps.signalarc.com/schemas-index.html.

[10] *Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron*. ISO/IEC, 2016. ISO/IEC 19757-3:2016

[11] *CREPDL·Schematron*. Available at: https://github.com/signalarc/crepdl-schematron.sch

[12] *FontCheck tool*. Available at: https://apps.signalarc.com/fontcheck/.

[13] *Document Schema Definition Languages (DSDL) Part 4: Namespace-based Validation Dispatching Language (NVDL)*. ISO/IEC, 2006. ISO/IEC 19757-4:2006

[14] *Document Schema Definition Languages (DSDL) Part 8: Document Schema Renaming Language (DSRL)*. ISO/IEC, 2008. ISO/IEC 19757-8:2008

[15] *Date and time — Representations for information interchange Part 2: Extensions*. ISO, 2019. ISO 8601-2:2019

[16] *Evaluation and Report Language (EARL)*. W3C, 2017. Available at: https://www.w3.org/TR/EARL10-Schema/

[17] *Syntactic metalanguage — Extended BNF*. ISO/IEC, 1996. ISO/IEC 14977:1996

XProc-Baseline

Designing a Portable Regression Testing Library for XProc Pipelines

Tomos Hillman
Oxford University Press
<tomos.hillman@oup.com>

Abstract

Real-world XProc pipelines do far more than transform data. They orchestrate file operations, create deliverables, manage archives, and produce outputs with non-deterministic content. This paper identifies the need for an automated regression testing framework for XProc pipelines and file-based workflows, in addition to complementary testing libraries such as XSpec.

We discuss the core challenges of testing file and archive management, configurable canonicalization of non-deterministic content, and integration with modern CI/CD platforms. We present our approach to building XProc-Baseline: a portable, reusable library that addresses the gaps in current XML testing tools by providing configurable canonicalization, manifest-based comparison, and seamless CI/CD integration.

Keywords: XProc, Testing, Continuous Integration

1. Problem Statement

XProc pipelines in production environments handle far more than data transformation [2]. They may manage file operations, create deliverable packages, handle nested archives, and orchestrate complex workflows [3]. Testing these systems end-to-end has proven difficult with existing tools, leading many to rely on manual testing for their most critical outputs.

1.1. Current Challenges

File and Archive Management

XProc pipelines manage ZIP creation, recursive folder structures, and nested archives. Validating that files are correctly packaged, directory structures are preserved, and archives are properly formed requires comparing complex file hierarchies and binary outputs. Directly comparing files with non-deterministic content (timestamps, UUIDs)

	will fail, leaving many engineers to rely on manual comparison.
Non-Deterministic Content	Output from production pipelines frequently use generated values: timestamps indicating when a delivery was created, UUIDs for tracking, automatically generated identifiers, and other metadata that changes with each execution. Regression testing requires comparing outputs despite these changes, making direct binary or text comparison impossible without sophisticated canonicalization.
Configurable Canonicalization	Different projects have different needs for what should be stripped, normalized, or transformed before comparison. A general-purpose library must support configurable canonicalization rules that can be customized per-project (or possibly per-test) without requiring extensive code changes.
Engine Portability	Regression tests must run in the same engines ([7], [8]) that production pipelines use. A reusable library must work across different XProc engines, XSLT processors, and projects without tight coupling to specific tools or platforms.
CI/CD Integration	Integrating regression tests into automated build pipelines requires machine-readable output, clear reporting, and seamless integration with version control systems. Current approaches are often project-specific and difficult to share across teams. Standard (or de facto standard) report formats such as JUnit XML [11] can support integration with existing CI infrastructure.

1.2. Relationship to Existing XML Testing Approaches

Mature tools already exist for testing XML processing logic, particularly XSpec for testing XSLT, and XQuery [9]. XSpec is also in the process of developing testing capabilities for XProc [10].

The author is a strong proponent of using XSpec for test driven development, writing tests to ensure that code requirements are understood before code is

written, and fulfilled once it is. Frameworks such as XSpec are vital for this sort of unit testing.

XSpec can also be used to perform some simple regression testing; however, XProc pipelines often operate at a broader level: managing files, archives, and multi-step workflows that produce complex deliverable packages. In these scenarios, the challenge is not only verifying individual transformations but validating the resulting file sets and artifacts as a whole. XProc-Baseline is intended to complement existing XML testing approaches by addressing this workflow- and artifact-level testing problem.

1.3. Impact

Without automated regression testing at the pipeline level, teams either skip end-to-end testing entirely or maintain brittle, project-specific test scripts. This leads to:

- Manual testing workflows that don't scale with pipeline complexity
- High maintenance burden for project-specific test code
- Difficulty adopting automated CI/CD systems
- No widely adopted "best practice" across organisations

2. Proposed Approach: XProc-Baseline

We propose XProc-Baseline, a framework for automating regression testing of XProc pipelines and file-based workflows. Rather than comparing binary outputs directly, XProc-Baseline uses manifest-based comparison with configurable canonicalization to handle non-deterministic content. The framework is designed to be portable, reusable, and suitable for integration into modern CI/CD platforms.

2.1. Core Concepts

2.1.1. Testing as a specification

Testing for a project will be defined using a specification file format. This format can then be used to generate a testing pipeline. The format may define:

- The pipeline to be tested
- Canonicalization process (as a pipeline step)
- Input sources for each test
- The baseline, or expected output from the file format, for each test.

2.1.2. Manifest-Based Comparison

Instead of comparing raw ZIP files or file trees, XProc-Baseline generates *canonicalized manifests* that describe the structure and content of (canonicalized) outputs:

- File paths and directory structures
- Content hashes of file contents where possible
- Metadata (size, MIME type, etc.)

This approach means that comparisons can be quick and meaningful. Manifests are text-based and version-control friendly, enabling code review of baseline changes and clear audit trails.

2.1.3. Configurable Canonicalization

A canonicalization framework that allows teams to define rules for handling non-deterministic content:

- Strip or remove fields entirely (timestamps, UUIDs, generated IDs)
- Normalize fields using pattern matching and replacement
- Handle both XML and binary content
- Support per-project and per-test customization

Custom canonicalization is supported by providing XProc steps by QName.

2.1.4. Engine-Agnostic Design

Built using standard XProc 3.x [2], XProc-Baseline is designed to work across:

- Different XProc engines ([7], [8])
- Different projects and organizations without modification

This portability is essential for adoption: teams can integrate XProc-Baseline into existing pipelines without replacing their current toolchain.

2.1.5. CI/CD Integration

XProc-Baseline is designed for seamless integration into modern CI/CD platforms:

- Provides JUnit test reports for easy integration with CI systems [11]
- Generates machine-readable manifests for easy parsing by CI tools
- Supports baseline versioning in Git with clear change tracking

2.1.6. Baseline Management and Caching

XProc-Baseline leverages version control for baseline management:

- Baselines stored in version control alongside test inputs.
- Manifest files may also be cached in version control
- Content hashes enable efficient change detection
- Git history provides audit trail of intentional baseline updates

This approach enables code review of baseline changes: when a test output changes, reviewers can see exactly what changed and decide whether the change is intentional or indicates a regression.

3. Architecture and Design

XProc-Baseline's architecture addresses the core challenge of regression testing without relying on dynamic step evaluation in XProc, which may not be available across implementations. Instead, the framework creates a test harness that orchestrates pipeline testing, manifest generation, and baseline comparison.

3.1. Overview

XProc-Baseline testing follows the following distinct phases:

1. *Test Configuration & baseline manifest generation*: Regression tests are configured and saved as a configuration file. This file includes any XProc libraries that are needed for testing and/or canonicalization, and defines the baseline results that are required for comparison.
2. *Test Harness Generation*: Given a regression test configuration file, a test harness XProc pipeline is generated. This harness imports all required libraries (the baseline library, the pipeline under test, and custom canonicalization steps) and orchestrates the testing workflow.
3. *Baseline Manifest Generation*: Baseline manifests are (re)generated and are stored in/referenced by this config file when the test harness is generated, and each time the expected output or canoncialization changes. Otherwise this phase is redundant.
4. *Manifest Generation and Canonicalization*: The test harness runs the pipeline under test, then generates a canonicalized manifest from its output.
5. *Manifest Comparison and Reporting*: The generated manifest is compared against the stored baseline manifest. Differences are reported and structured as JUnit output for CI/CD integration, and/or as HTML for human inspection.

This model trades the complexity of dynamic evaluation for the simplicity of static code generation — a pragmatic choice that improves portability and maintainability.

3.2. Detailed Phase Descriptions

3.2.1. Test Configuration Format

XProc-Baseline tests are defined using a declarative XML configuration file. This file specifies the pipelines to test, the inputs and options for each test, the expected baseline outputs, and project-specific canonicalization rules. By separating test definition from test execution logic, the configuration remains readable, maintainable, and independent of the underlying XProc implementation.

The format is defined in RelaxNG Compact, which can be found at `src/main/schema/xproc-baseline.rnc` in the project repository [1].

3.2.1.1. Configuration File Structure

A typical regression test configuration file has the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<b:regression-tests
  xmlns:b="http://ns.oup.com/xproc/baseline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <!-- Libraries to import for testing -->
  <b:imports>
    <b:import href="lib/my-pipeline.xpl"/>
    <b:import href="lib/my-canonicalization.xpl"/>
  </b:imports>

  <!-- Global configuration -->
  <b:config>
    <b:test-harness href="build/test-harness.xpl"/>
    <b:hash algorithm="crc"/>
    <b:canonicalization pipeline="my:canonicalize"/>
  </b:config>

  <!-- Individual test cases -->
  <b:test pipeline="my:my-pipeline" xml:id="test-001">
    <b:description>Process simple XML file</b:description>
    <b:options>
      <b:option name="output-format" select="'xml'"/>
      <b:option name="validate" select="true()"/>
    </b:options>
    <b:input port="source">
      <b:document href="input/document.xml"/>
    </b:input>
    <b:output uri="build/output/test-001/"/>
    <b:baseline uri="baseline/test-001.xml"/>
```

```
    </b:test>

    <b:test pipeline="my:my-pipeline" xml:id="test-002">
      <b:description>Process ZIP archive</b:description>
      <b:config>
        <!-- override canonicalization for this specific test -->
        <b:canonicalization pipeline="my:canonicalizeSpecial"/>
      </b:config>
      <b:options>
        <b:option name="output-format" select="'csv'"/>
      </b:options>
      <b:input port="source">
        <b:document href="input/document.xml"/>
      </b:input>
      <b:output uri="build/output/test-002/"/>
      <b:baseline>
        <b:manifest>
          <b:file name="output.xml" hash="fe84e1af"/>
          <b:file name="data.csv" hash="140c241e"/>
        </b:manifest>
      </b:baseline>
    </b:test>

  </b:regression-tests>
```

The root element is `b:regression-tests` in the namespace `http://ns.oup.com/xproc/baseline`. It contains four main child elements:

`b:imports`	Declares one or more XProc libraries to import. These typically include the pipeline(s) under test and any custom canonicalization libraries. This element is required.
`b:config`	Specifies global configuration options that apply to all tests unless overridden at the test level. See Section 3.2.1.2 below.
`b:test` (one or more)	Defines an individual test case. See Section 3.2.1.3 below.

3.2.1.2. Global Configuration (`b:config`)

The `b:config` element at the root level establishes defaults for all tests. It may contain:

`b:test-harness`	Required. Specifies the URL of the harness pipeline that needs to be generated from the test definition.

`b:hash` — Specifies the hash algorithm used for computing content hashes in manifests. Valid algorithms are:

- `md5` — MD5 hash (32 hex characters)
- `sha` — SHA-1 hash (40 hex characters)
- `crc` — CRC checksum (8 hex characters, faster but less reliable)

Default: `crc`. The attribute syntax is `algorithm="crc"`.

`b:canonicalization` — Specifies the default canonicalization step to apply during manifest generation. The `@pipeline` attribute names the step by its QName (e.g., `my:canonicalize`). This step will be invoked for each entry in the manifest before hashing (see Section 3.2.3.1).

If no canonicalization is desired, omit this element.

3.2.1.3. Individual Test Cases (`b:test`)

Each `b:test` element defines a single regression test. It has two required attributes and several child elements:

`@pipeline` (required) — The QName of the pipeline step to test (e.g., `my:my-pipeline`). This pipeline must be defined in one of the imported libraries specified in `b:imports`.

`@xml:id` (required) — A unique identifier for this test within the configuration file (e.g., `test-001`). Used in reporting and to correlate test results with baselines.

Child elements of `b:test`:

`b:description` — A human-readable description of what this test does. Used in reports and documentation. Required.

`b:config` — Test-level configuration, overriding the global config. Currently only supports `b:canonicalization`. Optional; if omitted, the global config applies.

`b:options` — Options to pass to the pipeline under test. Contains one or more `b:option` child elements, which have a corresponding model to XProc `p:with-option` elements. Optional.

`b:input` — Specifies input data for the pipeline. The data model here corresponds to `p:with-input` for providing input options to pipeline steps. Optional.

`b:output` — Specifies output data from the pipeline. Required.

`b:baseline` — Specifies the expected output by URI. Required.

The baseline manifest may be provided, either as an embedded manifest or as a reference to an external; this is optional in the schema, but must be provided when running the test-harness; see Section 3.2.3.

When referencing an external XML file, use the `@manifest-href` attribute:

```
<b:baseline uri="..." manifest-
href="baseline/test-001.xml"/>
```

The URI is resolved relative to the location of the regression test configuration file. This approach is recommended for complex or large outputs, as it keeps the configuration file readable and enables baseline reuse across multiple tests.

Alternatively, the manifest may be embedded directly:

```
<b:baseline>
  <b:manifest>
    <b:file name="output.xml" content-
type="application/xml"
            size="1024"
hash="abc123def456..."/>
    <b:file name="data.csv" content-
type="text/csv"
            size="512"
hash="fed654cba321..."/>
  </b:manifest>
</b:baseline>
```

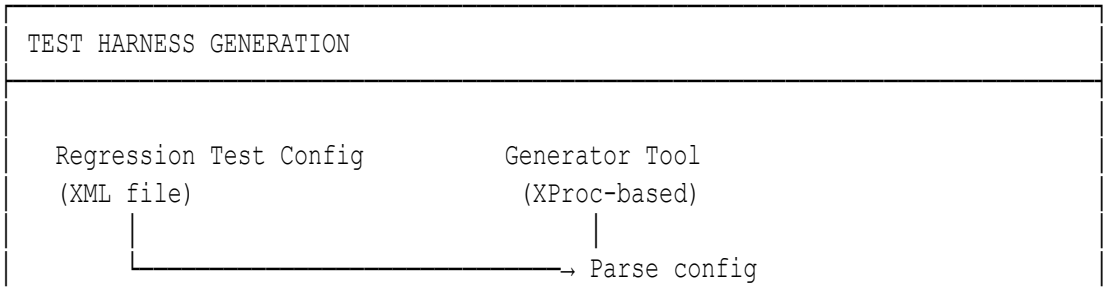
This approach is suitable for simple tests with few or small outputs. The manifest structure is described in Appendix A.

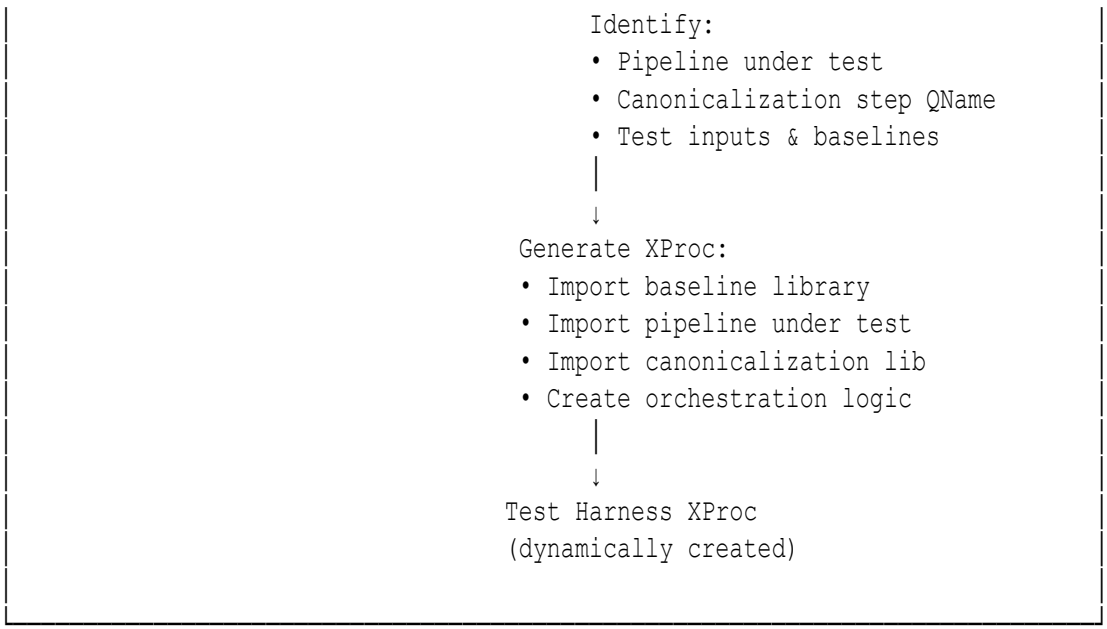
3.2.2. Test Harness Generation

The test harness is a generated XProc pipeline that serves as the entry point for both testing and baseline manifest generation. It is created by the generator step of the Baseline library which:

1. Parses the regression test configuration
2. Extracts key metadata: the pipeline under test (by QName), the canonicalization step to invoke (by QName), and the baseline manifest location (by `@href` or inline)
3. Generates a new XProc document that declares the required `p:import` statements for:
   - The baseline library itself (providing manifest generation steps)
   - The pipeline under test
   - Any custom canonicalization libraries
4. Construct a "generate" orchestration pipeline that will:
   - Parse the regression test configuration
   - Pass each baseline definition to the manifest generation
   - Invoke the canonicalization step by its resolved QName
   - Store the result of the baseline manifest generation at the location specified by `@manifest-href` if present, or as an embedded element if not.
5. Constructs a "testing" orchestration pipeline that will:
   - Receive test inputs
   - Invoke the pipeline under test with configured options
   - Pass the output to manifest generation
   - Invoke the canonicalization step by its resolved QName
   - Compare the result against the baseline
   - Report outcomes

By generating the harness at this stage, we avoid the need for dynamic step evaluation (`p:run` or equivalent). All imports are statically resolvable at compile time, ensuring consistent behavior across XProc engines.



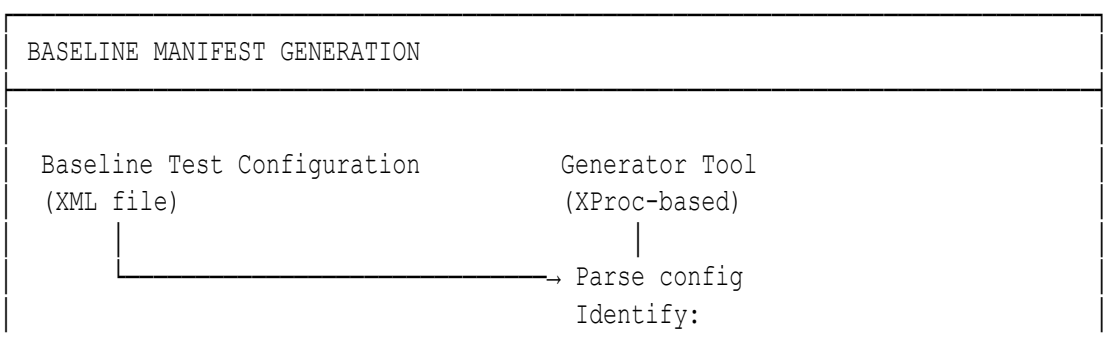


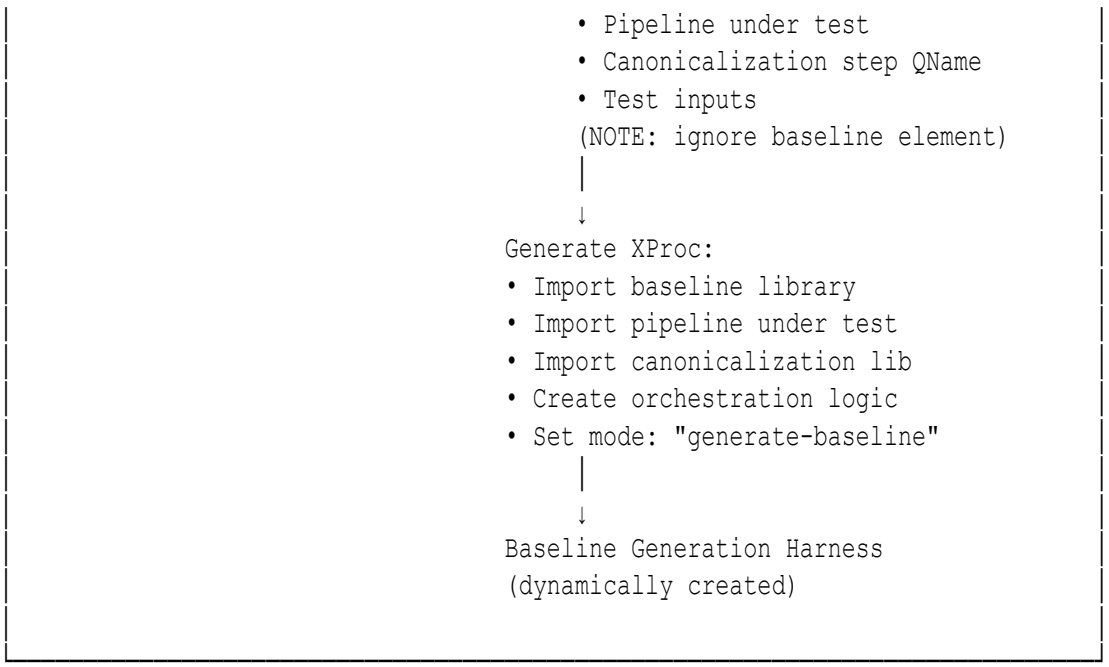
Key Design Decision: The baseline library defines placeholder steps, e.g. the canonicalisation step. These placeholders are never intended to be executed; instead, the test harness generator ensures that the actual canonicalization step (identified from the config) is imported and will be invoked by its QName during manifest generation. This allows the baseline library to remain engine-agnostic and project-independent, while the harness provides project-specific logic.

3.2.3. Baseline Manifest Generation and Canonicalization

Baselines are typically generated by running the pipeline once with known-good inputs and capturing the manifest output. The generated manifest is then reviewed, stored in version control, and used as the baseline for future test runs.

To generate a baseline manifest, the test harness is run using the "generate" step, which produces the manifest without comparison. The output is then saved and may be committed to the repository alongside or as part of the test configuration. See Appendix A for further details.





3.2.3.1. Integrated Canonicalization

During manifest generation, canonicalization is applied to each entry in place. Rather than post-processing the entire manifest, canonicalization rules are invoked as part of the manifest creation process:

1. For each file or archive entry, before hashing, the test harness invokes the canonicalization step identified in the test configuration (by QName)
2. The canonicalization step receives the entry and returns a canonicalized version
3. The hash is computed on the canonicalized content

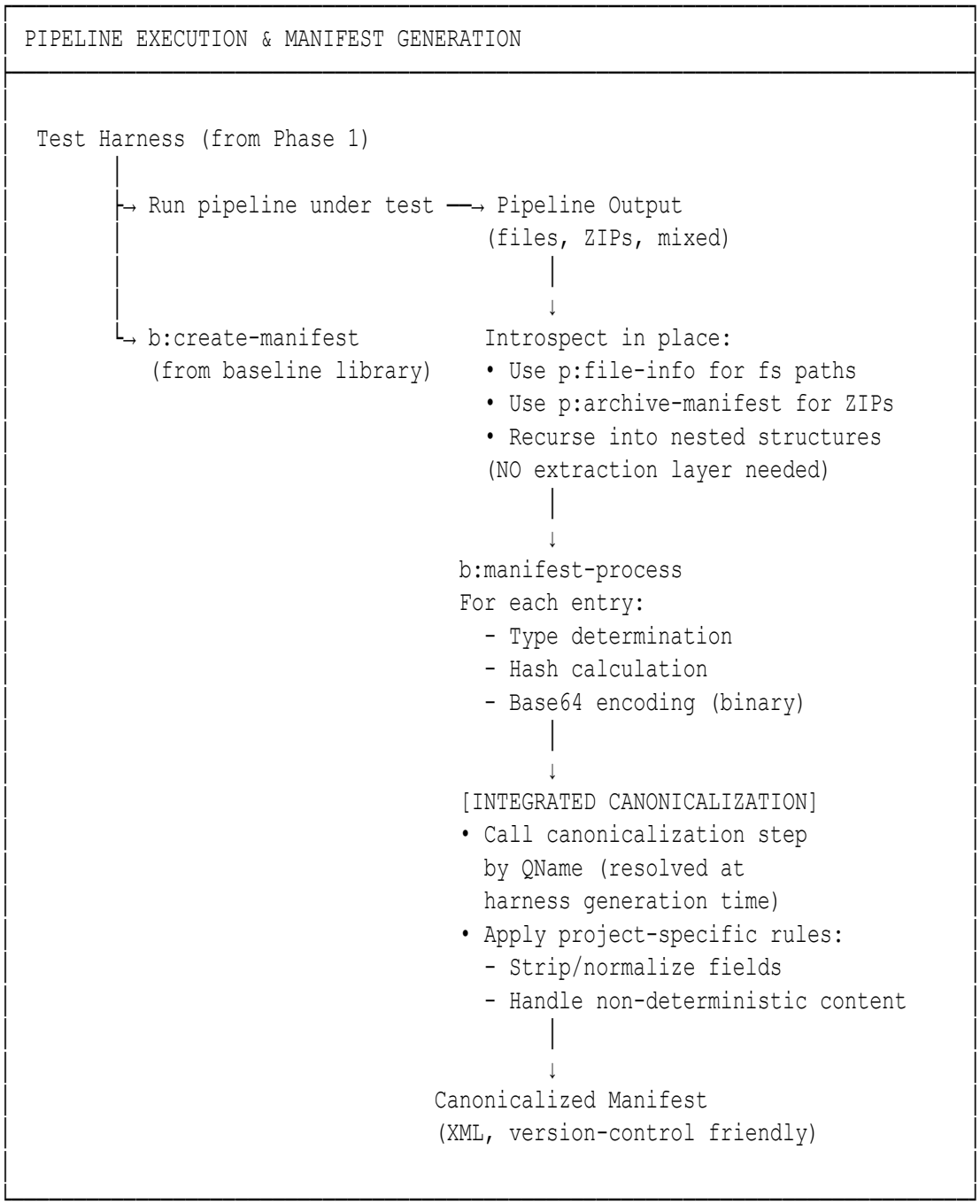
This integration allows project-specific rules to be applied uniformly across all outputs:

Strip non-deterministic fields	Remove `<timestamp>`, `<uuid>`, or other auto-generated elements from XML files before hashing
Normalize patterns	Apply regex replacement rules to version strings, file paths, or identifiers that may vary between runs

Implementation Detail: The baseline library defines a placeholder canonicalization step that is never used at runtime. Instead, the test harness generator imports and resolves the actual canonicalization step by QName, ensuring that the baseline library itself remains independent of project-specific logic.

3.2.4. Running the test Harness

Once the harness is generated, it can be executed using the "testing" step, producing the output for the pipeline(s) under test. XProc-Baseline then generates a manifest for each test by canonicalizating the output.



3.2.4.1. In-Place Introspection

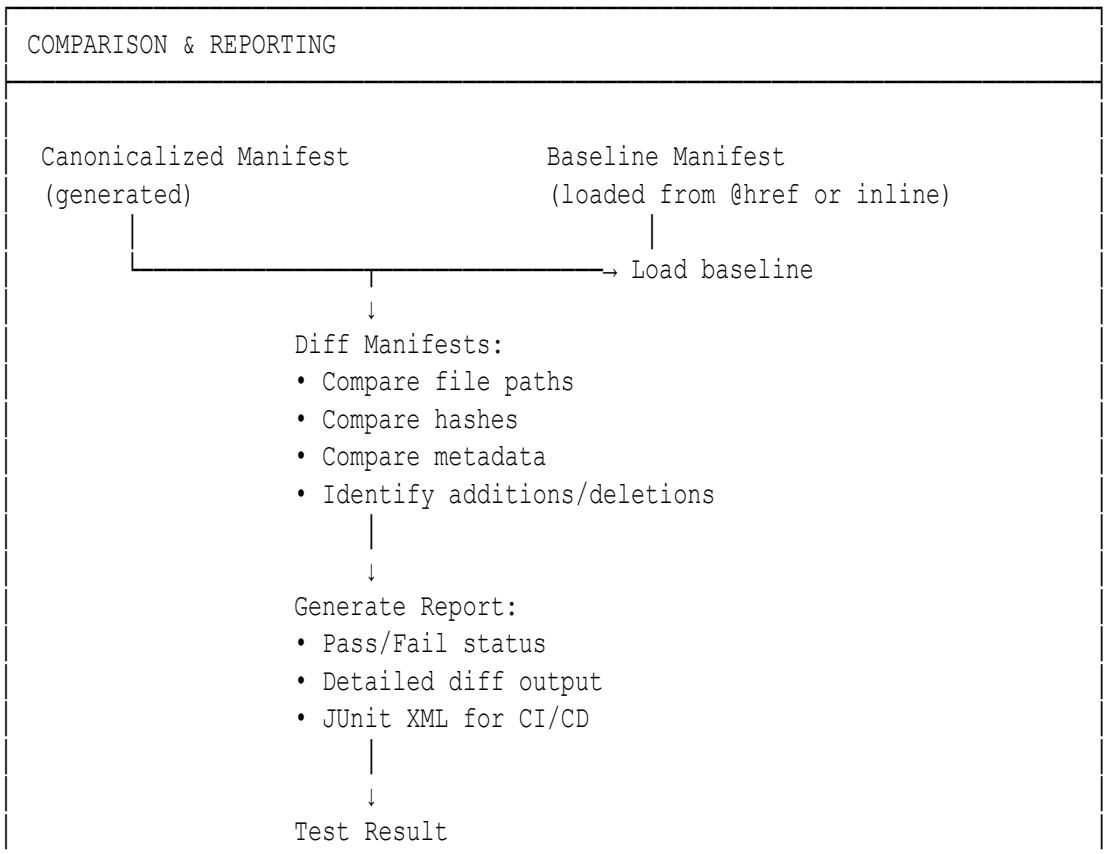
The manifest generation process uses XProc's built-in steps to examine file and archive structures directly:

- `p:file-info` examines filesystem paths and provides metadata (size, modification date, content type)
- `p:archive-manifest` lists the contents of ZIP archives without decompressing them
- `p:unarchive` selectively extracts individual entries for hashing, then re-archives the result (or discards it)
- Recursive invocation via `p:viewport` allows nested archives to be processed depth-first

This approach eliminates the need for a separate extraction layer and reduces temporary file overhead, especially for large or deeply nested archives.

3.2.5. Comparison and Reporting

After manifest generation and canonicalization, the generated manifest is compared against the baseline manifest.



3.2.5.1. Baseline Manifest Loading

The baseline manifest can be provided in two ways (specified in the test configuration):

External file (via @manifest-href)	The baseline manifest is stored in a separate XML file, referenced by relative or absolute URI. This approach is recommended for large or complex outputs, as it keeps the test configuration readable and enables baseline reuse across multiple tests.
Inline	The baseline manifest is embedded directly within the test configuration element. This approach is useful for simple tests where the expected output is small and self-documentation is beneficial.

The framework does not mandate co-location. Baseline manifests can be stored alongside test configurations, with source data, or in a dedicated baseline directory— users choose the organization that best suits their project.

3.2.5.2. Diff and Comparison

Once both manifests are loaded, they are compared as XML trees. The comparison checks:

1. *Structural equivalence:* File paths, directory hierarchies, and archive nesting must match
2. *Hash matching:* Content hashes must be identical
3. *Metadata consistency:* Size, content-type, and compression metadata should be consistent.
4. *Additions/deletions:* New files in the generated output or missing files from the baseline are flagged

3.2.5.3. Test Reporting

Test outcomes will be reported in two formats:

JUnit XML	A standardized XML report suitable for CI/CD systems (Jenkins, GitHub Actions, GitLab CI, etc.). Each test case is represented with pass/fail status and detailed failure messages.

Human-Readable Diff	A structured diff showing which files were added, removed, or modified (hash mismatch). line-by-line comparisons can be fetched from corresponding files in the case of non-binary file mismatches. This output aids in debugging and in code review when baseline changes are intentional.

Both formats enable teams to understand test failures and make informed decisions about whether to accept baseline changes or fix the pipeline.

3.3. Key Design Decisions and Rationale

3.3.1. Static Test Harness Generation vs. Dynamic Evaluation

Rather than using dynamic XProc step evaluation (`p:run`), XProc-Baseline generates a static test harness at configuration time. This choice:

- *Improves portability:* All imports are resolved at generation time, avoiding inconsistencies in dynamic evaluation across Calabash, Morgana, and other engines
- *Simplifies debugging:* The generated harness is an actual XProc file that can be inspected, logged, and tested in isolation
- *Enables compile-time validation:* Schema validation and step resolution occur during harness generation, catching errors early

3.3.2. In-Place Manifest Generation vs. Extraction

We originally considered an extraction layer that would decompress files to a temporary directory. This has been replaced with in-place introspection:

- *Performance:* Avoids the overhead of writing and reading temporary files, especially critical for multi-gigabyte archives
- *Simplicity:* Fewer intermediate steps means fewer error points and simpler orchestration
- *Scalability:* Streaming-compatible design enables future optimization for very large outputs

3.3.3. Canonicalization as an Integrated Step

Canonicalization is applied during manifest generation, not as a post-processing step on the entire manifest. This approach:

- *Preserves baseline simplicity:* The baseline library remains project-independent; canonicalization rules are injected at harness generation time

- *Reduces manifest size:* Non-deterministic content is removed before hashing, not stored and then ignored during comparison
- *Enables entry-level customization:* Different canonicalization rules can be applied to different file types or entries as needed

3.3.4. Base64 Encoding for Binary Content Hashing

Binary files are base64-encoded before hashing. While this adds a small computational overhead, it provides significant benefits:

- *Version control friendly:* Manifests remain pure XML and can be stored in Git alongside test data
- *Portable:* Base64 encoding ensures consistent results across platforms and XProc engines
- *Deterministic:* The same binary file always produces the same base64 string and hash, independent of filesystem or execution environment

3.4. Open Design Questions and Future Work

3.4.1. Human Readable report format

Currently we only have JUnit test results, which are fine for supporting simple pass/fail results for supporting e.g. Continuous Integration. Human developers need a more accessible report which identifies which files/structures are changed, provided with a diff of either the manifest differences at a minimum, but preferably with line by line diffs within those files. This will enable speedier resolution of test failues.

3.4.2. Handling of non-filesystem results

Pipeline steps which produce direct outputs on the result port currently have no way of being tested; how XProc Baseline should handle such cases is an open question.

One possibility is to store the results from the output port(s) at the URI specified by `b:output`.

3.4.3. Community Engagement

Currently XProc Baseline is a single developer effort; to be demonstrably reliable, it is hoped that it will eventually be adopted by other XProc users, and by other XProc developers to meet their needs.

There is a guide for contributing on GitHub [1], and pull requests and issues are welcome.

3.4.4. Streaming Comparison for Very Large Manifests

The current comparison implementation loads both manifests into memory. For outputs larger than available RAM, a streaming or lazy-evaluation approach would be beneficial. This is deferred to a future optimization phase if performance testing identifies it as a bottleneck.

3.4.5. Metadata Strictness in Comparison

The comparison currently verifies metadata (size, content-type, compression method) in addition to content hashes. The degree of strictness may be configurable in future releases, allowing teams to choose whether metadata mismatches are warnings or failures.

References

XProc Baseline on GitHub

[1] *XProc-Baseline: A Library for XProc Regression Testing*. . 2026-03-31. https://github.com/OUP2/XProc-Baseline . Copyright © 2026 Oxford University Press.

XML Standards and Specifications

[2] *XProc 3.1: An XML Pipeline Language*. Norman Walsh, Achim Berndzen, Gerrit Imsieke, and Erik Siegel. XProc Next Community Group. World Wide Web Consortium (W3C). https://spec.xproc.org/3.1/xproc/ .

[3] *XProc 3.1: Standard Step Library*. Norman Walsh, Achim Berndzen, Gerrit Imsieke, and Erik Siegel. XProc Next Community Group. World Wide Web Consortium (W3C). https://spec.xproc.org/3.1/steps/ .

[4] *XProc 3.1: Pipeline Execution*. Norman Walsh, Achim Berndzen, Gerrit Imsieke, and Erik Siegel. XProc Next Community Group. World Wide Web Consortium (W3C). https://spec.xproc.org/3.1/run/ .

[5] *XSL Transformations (XSLT) Version 3.0*. World Wide Web Consortium (W3C). https://www.w3.org/TR/xslt-30/ .

XProc Implementations and Tools

[6] *XProc.org*. XProc Community. https://xproc.org .

[7] *XML Calabash*. Norman Walsh. https://www.xmlcalabash.com .

[8] *MorganaXProc-III*. Achim Berndzen. XML Project. https://www.xml-project.com/morganaxproc-iii.html .

Testing and Continuous Integration

[9] *XSpec: Unit Testing Framework for XSLT, XQuery, and XProc*. XSpec Project. https://xspec.io/ .

[10] *To XProc Users: Welcome to XSpec*. Amanda Galtman. *xspectacles blog* . https://medium.com/@xspectacles/to-xproc-users-welcome-to-xspec-6448a032e2e5 .

[11] *JUnit XML Format*. https://github.com/testmoapp/junitxml .

A. Manifest Format (`b:manifest`)

A manifest describes the structure and content of pipeline outputs. It is an XML document with a tree structure mirroring the filesystem or archive hierarchy being tested.

A.1. Manifest Structure

The manifest root is a single structural element—either a `b:file`, a `b:folder`, or a `b:archive`—depending on the output type:

```
<!-- Output is a single file -->
<b:manifest>
  <b:file name="output.xml" content-type="application/xml"
          size="1024" hash="abc123..."/>
</b:manifest>

<!-- Output is a directory -->
<b:manifest>
  <b:folder name="output">
    <b:file name="file1.xml" content-type="application/xml"
            size="512" hash="def456..."/>
    <b:file name="file2.txt" content-type="text/plain"
            size="256" hash="ghi789..."/>
    <b:folder name="subdir">
      <b:file name="nested.xml" content-type="application/xml"
              size="128" hash="jkl012..."/>
    </b:folder>
  </b:folder>
</b:manifest>

<!-- Output is a ZIP archive -->
<b:manifest>
  <b:archive name="output.zip" content-type="application/zip"
             size="2048" compression-method="deflate">
    <b:file name="file1.xml" content-type="application/xml"
            size="512" hash="def456..."/>
    <b:folder name="subdir">
      <b:file name="nested.xml" content-type="application/xml"
              size="128" hash="jkl012..."/>
    </b:folder>
```

```
    </b:archive>
  </b:manifest>
```

A.2. Element Types

`b:file`	Represents a single file (leaf node). No child elements.	
	`@name` (required)	The filename or path segment.
	`@content-type` (optional)	MIME type of the file (e.g., `application/xml`, `text/plain`, `application/pdf`).
	`@size` (optional)	File size in bytes. Used for metadata verification.
	`@hash` (required)	Content hash computed according to the configured hash algorithm. For binary files, the hash is computed on the base64-encoded content.
`b:folder`	Represents a directory. May contain `b:file`, `b:folder`, or `b:archive` child elements.	
	`@name` (required)	The directory name or path segment.
`b:archive`	Represents a ZIP archive. May contain `b:file`, `b:folder`, or nested `b:archive` child elements.	
	`@name` (required)	The archive filename.
	`@content-type` (optional)	Should be `application/zip` for ZIP files.
	`@size` (optional)	Archive size in bytes.
	`@compression-method` (optional)	Compression algorithm for entries (typically `deflate` or `store`).

A.3. Manifest Structure and Content Hashing

The manifest is an XML document describing the output structure. Each entry includes:

Name	File or directory path relative to the output root
Content-Type	MIME type (e.g., `application/zip`, `application/xml`, `application/pdf`)
Size	File size in bytes (for metadata verification)

Hash	Content hash (algorithm configurable: MD5, SHA-1, CRC, etc.) computed as follows: • *XML files:* Hash the serialized XML representation (with stable element/attribute ordering where possible) • *Binary files:* Base64-encode the binary content, then hash the resulting string. This ensures portability and version-control compatibility when manifests are stored in XML. • *Archives:* Recursively hash contents using the same rules
Compression-Method	For archive entries, the compression algorithm (e.g., `deflate`, `store`)

The manifest is text-based, XML-formatted, and therefore:

- Easily diff-able in version control
- Reviewable by humans and automated systems alike
- Suitable for long-term archival alongside test data

A.4. Manifest Example: Complex Output

Here is a realistic example of a manifest for a complex output containing files, directories, and a nested archive:

```
<b:manifest>
  <b:folder name=".">
    <b:file name="README.txt" content-type="text/plain"
            size="256" hash="a1b2c3d4e5f6..."/>
    <b:file name="config.xml" content-type="application/xml"
            size="512" hash="b2c3d4e5f6a7..."/>
    <b:folder name="data">
      <b:file name="input.csv" content-type="text/csv"
              size="1024" hash="c3d4e5f6a7b8..."/>
      <b:file name="processed.xml" content-type="application/xml"
              size="2048" hash="d4e5f6a7b8c9..."/>
    </b:folder>
    <b:archive name="results.zip" content-type="application/zip"
               size="4096" compression-method="deflate">
      <b:file name="summary.txt" content-type="text/plain"
              size="512" hash="e5f6a7b8c9d0..."/>
      <b:folder name="details">
        <b:file name="detail-1.xml" content-type="application/xml"
                size="1024" hash="f6a7b8c9d0e1..."/>
        <b:file name="detail-2.xml" content-type="application/xml"
```

```
                size="1024" hash="g7b8c9d0e1f2..."/>
      </b:folder>
    </b:archive>
  </b:folder>
</b:manifest>
```

The story of Gerald Cinamon's germandesigners.net

Transforming a complex MS Word manuscript for the web

Matt Patterson
Saxonica Limited
<matt@wekstatt.io>

Abstract

The renowned graphic designer and author Gerald Cinamon [1] wrote a comprehensive biographical dictionary about graphic designers working in Germany during the Nazi regime, 'German graphic designers during the Hitler period', a project well summarised in its author's introduction:

> *Design historians in America and Britain have tended to ignore the talents or lives of those who remained in Germany during the Nazi regime as being unworthy of attention. But the talents were there, and the lives went on. Simply because these designers lived in Nazi Germany is no reason to ignore their work. Any history of twentieth-century graphic design – and Germany's particularly – must take note of them and their work.*
>
> *Generally, the lives and work of most émigré designers (usually Jewish) have been covered in books and design journals. Here is featured, where known, what happened to the designers that remained during the period 1933 to 1945 – the Hitler period.*
>
> *—Gerald Cinamon, from the introduction to the German Designers site [3]*

The book was written as a manuscript in Microsoft Word, containing over 900 entries across several separate files.

When turning the manuscript into a viable print-published book seemed unlikely, I was approached about whether it would be possible to publish it as a website.

This is the story of how we were able to combine the author's Word files, metadata annotations in Word, and XSLT to create well-structured, richly annotated markup that could, in turn, be transformed into a work of Hypertext.

We'll cover:

- *How the source material was prepared in Word itself, and what techniques we used to annotate and enrich the text.*

- *How the Word files were processed into usable, structured, XML.*
- *How that XML was turned into a website.*

Keywords: XML, XSLT, HTML, JSON, Design history, MS Word

1. A brief history of the project

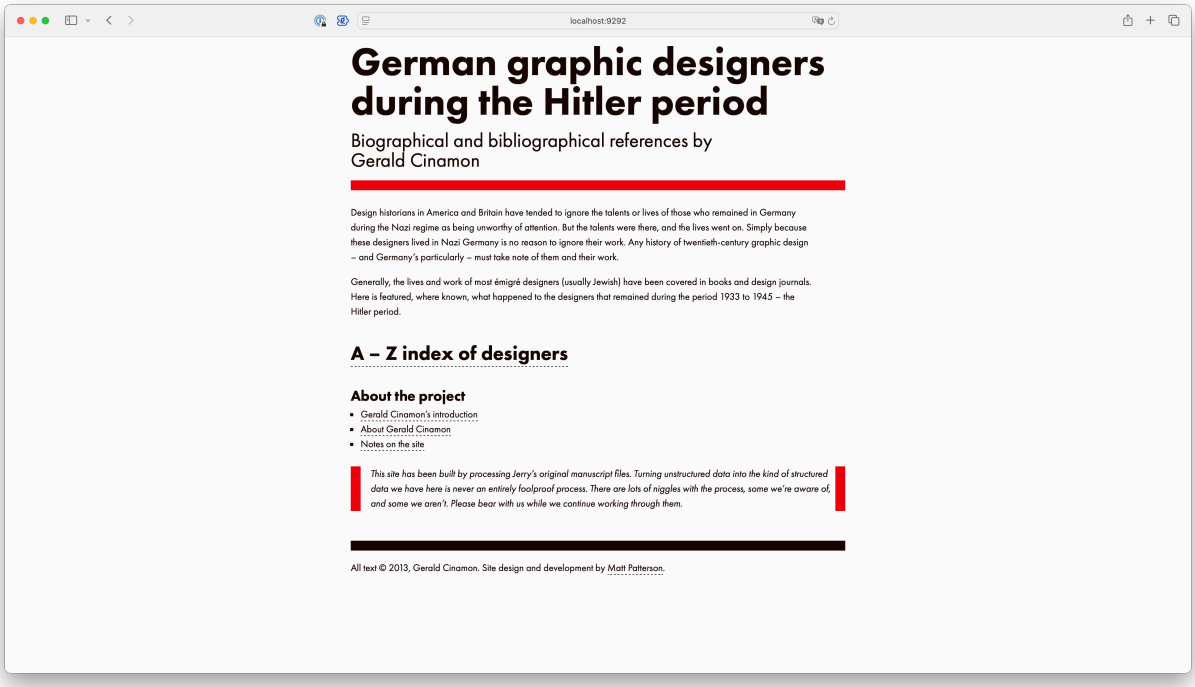


Figure 1. The home page

The project began back in 2011 with a call from the writer and academic Alex Butterworth about an unpublished biographical dictionary written by Gerald Cinamon. Gerald Cinamon, 1930-2024, is a significant figure in British Graphic Design and Typography – he was Penguin's Chief Designer, for example – who became a historian of the field.

I was intrigued, not least because Jerry was also the external examiner for my BA in Typography & Graphic Communication at Reading...

As a dictionary of biography, the finished book would be set of entries, one for each subject (person or organisation) being included. That format obviously lends itself to hypertext very well, and after much consideration and exploration we settled on a fairly straight hypertext adaptation approach, with one page per subject, and a central alphabetical index.

The book, still in author's manuscript form as a set of Microsoft Word documents, presented a number of challenges to convert into a coherent web site. Those challenges, and how we solved them, is the subject of this paper.

The original version of the site was a Ruby on Rails app, using XSLT 2 to perform basic extraction, but all the post processing in Ruby, with everything ending up in a relational database. The site went live to coincide with a major exhibition of Jerry's work in 2013 at the ICA in London [2]. Since then, the site has sat essentially unchanged until this year. Motivated by the need to move away from a largely unnecessary Web app server infrastructure, and the desire to make progress on features we'd had to abandon for the original site, we undertook to rewrite the machinery of the site from scratch, while preserving the URL structure and HTML output exactly. The preparation work done on the manuscript itself required no changes, but we are choosing to ignore the previous Rails site in this paper, for reasons of time and space.

This was a reasonably sized project, with a reasonably sized data set. The challenges are similar to those faced by people with bigger, and smaller, data sets, and more, or less, complex presentation requirements. It's a good size: big enough to have challenges, but small enough to be for the approaches we took to those challenges to be (hopefully) useful and understandable examples.

This paper will cover the preparation of the manuscripts, their extraction and processing, and the generation of a site from the processed results. Again, for reasons of time and space, we don't cover absolutely everything There will be things we elide, and very little discussion of the build system that glues the various parts of the process together. That said, all the examples are real, and all the code should run.

2. Preparing the source material in Word

Consisting of over 700 entries, some only a sentence, some several pages, the manuscript was split between a number of Word documents, largely to stave off crashes. Entries contained a variable-length biography, and could contain some or all of a bibliography of works referencing the subject, a bibliography of works by the subject, and a list of exhibitions about, or featuring the work of, the subject.

You can see a page from the original manuscript on Figure 2.

While Word documents can contain structure, it's inferred from paragraph styles, not made explicit through markup structure like nesting. The main questions for processing the Word documents are therefore:

- What structure is present within the file?
- How can it be extracted?
- What kinds of processing would this allow without editing the manuscript?

We are concerned, here, with the logical hierarchies of content within the manuscript, as they are expressed visually. That is, how are the parts of the document differentiated from each other, and how do those parts relate to each other as a hierarchy?

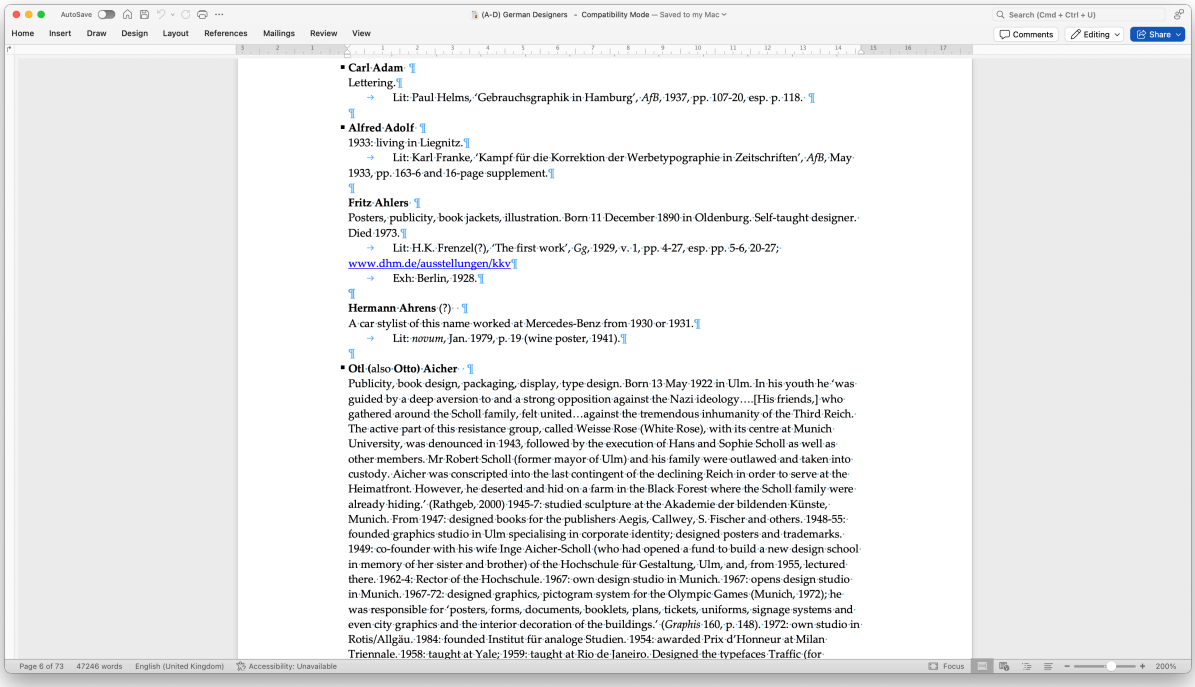


Figure 2. A page from the original manuscript

As we will discuss in the next section, the markup underlying a Word document is a flat sequence of paragraphs. It's possible to attach named paragraph styles to paragraphs, in which case the paragraph takes on the same visual appearance as all the other paragraphs with that style. With this approach it's possible to make some aspects of the underlying logical structure of the document explicit: this paragraph is body text, that one is a heading, and so on. It's also possible to directly apply visual formatting to the text without using styles (all paragraphs will have the 'Normal' style, but vary in visual formatting). Worse, you can directly apply visual formatting to text, overriding the appearance of, but not removing the association with, other styles. A paragraph may be marked as body text, but styled as a heading (or vice versa).

The editorial lever we have to make hierarchy explicit (as explicit as possible, anyway) in a Word document is the consistent application of paragraph and character styles. If that is done, it's possible to map those styles to another document model, or to imply grouping and nesting reliably. While Word styles are fundamentally limited by the Word document model (no more than one paragraph style per paragraph, and only one character style per character), they are the tool we have.

The manuscript, while visually consistent, with a clear visual hierarchy that maps to the author's underlying logical hierarchy, did that through direct formatting of the text. To transform the manuscript into another format required that we

develop, and consistently apply, a set of styles throughout the entire document set in order to make them transformable.

The work of devising a set of styles that balanced the competing needs of richness to allow complex transformations, and simplicity to allow consistent and reliable application of the styles to the documents by an editor made up the bulk of the project.

What we came up with was a set of simple paragraph styles for the components of an entry – name, the biography, the various sorts of bibliographic details, and a richer set of character styles that could be used to annotate the entry. There were styles to capture the various parts of bibliographic entries, as well as the general - a date, the name of another designer, and the specific: dates and places of birth and death.

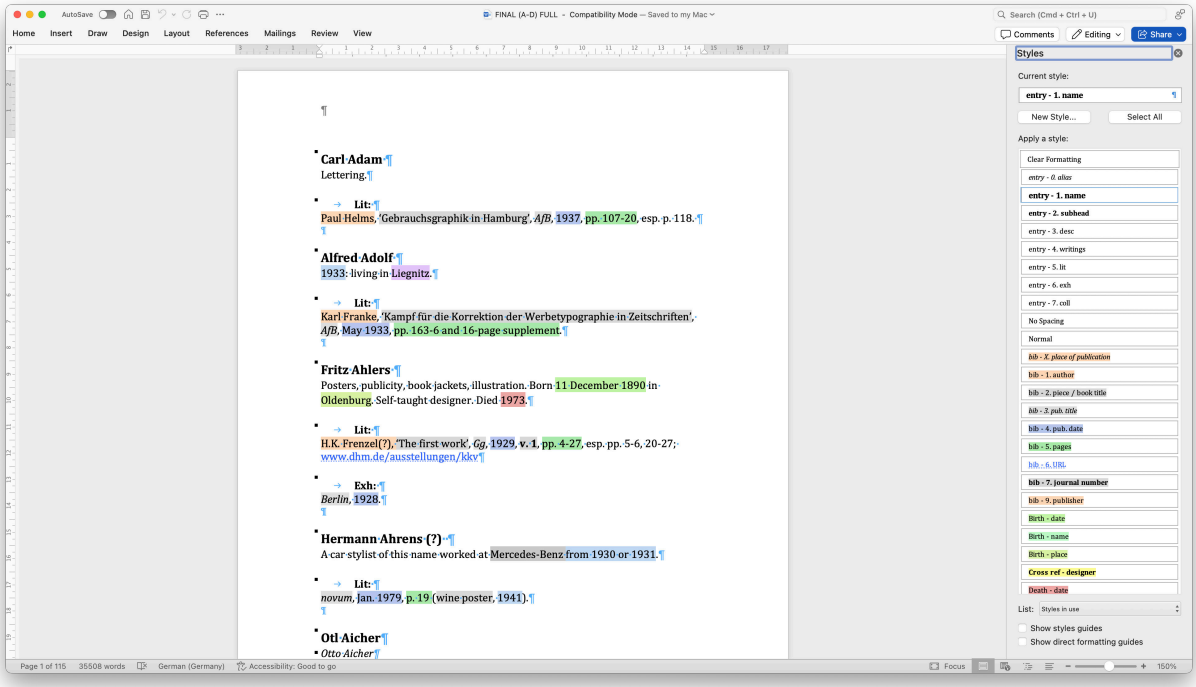


Figure 3. Short styled entries

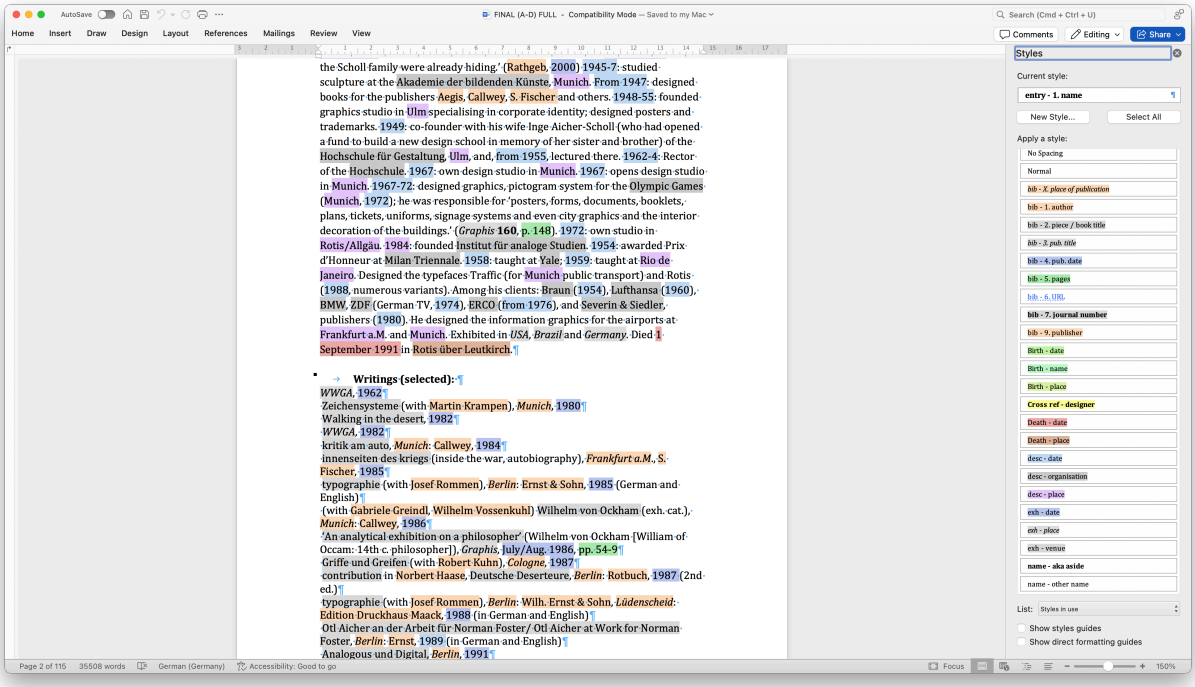


Figure 4. An excerpt of a longer entry

Once we had a set of styles that worked, we recruited someone to go through the whole document set and apply them. To aid that, the styles had a comprehensive set of keyboard shortcuts set up through macros to make it possible to do all the styling, without having to take your hands off the keyboard, an enormous productivity booster.

3. Turning one Word file into 100 dictionary entries

The primary aim of processing the Word documents is to extract the entries within them as separate documents. To do that we need to consider the structure of the WordprocessingML flavour of Microsoft's Office Open XML format, the affordances it offers us, and the restrictions it imposes. (We'll also look at some of the common pitfalls and gotchas I encountered.)

The format is standardised as ECMA-376 [7] & ISO/IEC 29500 [6], and there's a lot of documentation available. One thing is clear, though. It's an extremely complex format. This screen grab of the start of a WordprocessingML document should give you a flavour:

Figure 5. A WordprocessingML document

Critical things to note: First, that's an unholy number of namespace declarations. Second, namespace-qualified attribute names are used throughout, and third, the content is all one line. The line breaks in the screenshot above are only there because of oXygen's soft wrap. Let's look in more detail at the structure.

3.1. The structure of the WordprocessingML ML document

`.docx` files are actually just Zip archives with a different file extension. If we unzip a document at the command line you can see the result:

```
$ unzip ../a-word-file.docx
Archive:  ../a-word-file.docx
  inflating: [Content_Types].xml
  inflating: _rels/.rels
  inflating: word/_rels/document.xml.rels
  inflating: word/document.xml
  inflating: word/footnotes.xml
  inflating: word/endnotes.xml
  inflating: word/header1.xml
  inflating: word/footer1.xml
  inflating: word/theme/theme1.xml
  inflating: word/_rels/settings.xml.rels
  inflating: word/settings.xml
  inflating: word/fontTable.xml
```

```
  inflating: docProps/core.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: docProps/app.xml
```

Note that the Zip archive contains no root folder name as the archive, so unzip into a folder you have created for the task. Also note that, at least sometimes, macOS's built-in archive handling will refuse to unzip a `.docx` whose extension you have changed to `.zip`.

Depending on the complexity of your needs, you may need to look at several of these files, but if you're lucky, or otherwise careful you'll be able to do as I did, and ignore everything except the `word/document.xml` file. This file contains the main text of the document. If we strip the file back to its basic outline it looks like this:

```
<w:document>
  <w:body>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
    <w:p>
  </w:body>
</w:document>
```

The document is a flat sequence of `<w:p>` paragraph elements. Each paragraph, in turn, consists of some metadata followed by the text:

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="para-style"/>
  </w:pPr>
  <w:r>
    <w:t>The actual paragraph text!</w:t>
  </w:r>
</w:p>
```

There's often more to the `<w:pPr>` metadata, but for our purposes the `<w:pStyle>` element is all we need. Before we celebrate too early though, the text may well be a sequence of runs of text, with metadata of their own:

```
<w:p>
  <w:r>
    <w:t xml:space="preserve">The actual </w:t>
  </w:r>
```

```
    <w:r>
      <w:rPr>
        <w:rStyle w:val="character-style"/>
      </w:Pr>
      <w:t>paragraph text!</w:t>
    </w:r>
  </w:p>
```

This gives us two things: a handle on character-level markup, and the `xml:space` warning klaxon. Effectively, all whitespace in a document is significant, and careless processing, even serialising with indenting switched on, can wreak havoc. The text in a paragraph is not just broken into runs by changes in character styling, either. They can also be interrupted by control codes, like markers for the start or end of (alleged or real) spelling errors:

```
<w:p>
  <w:r>
    <w:t xml:space="preserve">The actual </w:t>
  </w:r>
  <w:proofErr w:type="spellStart"/>
  <w:r>
    <w:t>paragraph text!</w:t>
  </w:r>
</w:p>
```

There are many kinds of these empty tag markers that can be inserted by Word into your text flow, and there will be cases where text runs are split within a word to allow the placement of one of these. Evan Lenz has extensive coverage of all the things that can occur within WordprocessingML in his article about it [4]. If you want to find out more, this is an excellent resource.

3.2. The affordances of the WordprocessingML document

The document is flat sequence of paragraphs, which contain flat sequences of text runs. It's simple. The main affordance we are provided with is the `<w:pStyle>` and `<w:rStyle>` metadata tags which identify the style that is applied to a paragraph or text run. Using the presence of identifiers for styles we can view the styled manuscript like this:

```
<p entry-title>
<p>
<p>
<p>
<p entry-title>
<p>
<p bibliography-entry>
<p bibliography-entry>
```

```
<p bibliography-entry>
<p>
```

We can view the document as a series of entries delimited by the presence of a paragraph with the entry-title style, and, within those entries, sections like the bibliography represented by a sequence of adjacent paragraphs with a known style name:

```
<entry>
    <p entry-title>
    <p>
    <p>
    <p>
</entry>
<entry>
    <p entry-title>
    <p>
    <bibliography>
        <p bibliography-entry>
        <p bibliography-entry>
        <p bibliography-entry>
    </bibliography>
    <p>
</entry>
```

Within a paragraph, the principle is very similar, with the main wrinkle being the presence of marker elements breaking up runs of text:

```
<p>
    <r>They worked with the artist </r>
    <r crossref>Käthe </r>
    <spelling-marker/>
    <r crossref>Kollwitz</r>
    <r> in Berlin.</r>
</p>
```

If we squint, this resolves to:

```
<p>They worked with the artist <crossref>Käthe Kollwitz</crossref> in
Berlin.</p>
```

While it doesn't have the simplicity of gathering up directly adjacent elements, we can use similar techniques.

3.2.1. A brief digression into of style names in the UI, and IDs in the markup

The style ID in the `w:pStyle/@w:val` / `w:rStyle/@w:val` is just that – an ID. It's not the style name. Word has transformed the name you see in the UI, like "`entry`

`1. - name"`, to a token, `"entry1-name"`. The `.docx` also contains a `styles.xml` file that contains all styles, their UI names and their IDs:

```
<w:style w:type="paragraph" w:styleId="entry-1name">
    <w:name w:val="entry - 1. name"/>
    …
</w:style>
```

Within `styles.xml`, `<w:style w:styleId="…">` is the key the corresponds to `<w:pStyle w:val="…">` used in the document. You may find that you need to use this file to figure out which ID maps to a particular name style.

LibreOffice can also read and write `.docx` files, and, historically, LibreOffice / OpenOffice would use random opaque style IDs, and you really had to look up the ID in styles.xml to be sure of what style you were getting.

3.3. Splitting entries and grouping paragraph-level content

XSLT's `<xsl:for-each-group group-starting-with="…">` is exactly the tool for us. Using this, it's easy to chunk our document into entries and wrap them in a tag for later processing:

```
<xsl:for-each-group select="$body/*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-1name']">
    <entry>
        <xsl:sequence select="current-group()"/>
    </entry>
</xsl:for-each-group>
```

The expression given to `group-starting-with` is evaluated against the input sequence and if it matches the body of the `<xsl:for-each-group>` is executed with the `current-group()` function returning all the items that have been evaluated since the previous match.

The XPath selector we need is quite long, but it's also very simple. What it relies on, above all else, is consistent and correct styling of the document in Word. If one entry misses its subject name paragraph being styled correctly, you get two entries being merged, and assumptions that later processing steps make about the structure of the `<entry>` will be wrong, and if the input contained several non-entry items before the first paragraph styled `"entry1-name"` those would be returned as their own group. As well as this, because `group-starting-with` breaks at the beginning of a group, if the final group of items contains extra stuff that doesn't belong in an `<entry>`, the final `<entry>` would contain unexpected items.

In the case of this project, we worked hard to ensure that those sorts of problems were corrected in the source.

I'm able to use the same approach to subdivide the entry into chunks that align with entry sections because of the presence of subheadings between sec-

tions. The main difference is that significantly different behaviour is needed for the first chunk, which contains the name and biography of the entry subject, and doesn't contain a section subheading. I wrap each in a temporary <chunk> element and rely on being able to discriminate the first from following chunks to drive later processing:

```
<xsl:template match="entry">
    <xsl:copy>
        <xsl:for-each-group select="*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-2subhead']">
            <xsl:variable name="chunk">
                <chunk>
                    <xsl:sequence select="current-group()"/>
                </chunk>
            </xsl:variable>

            <xsl:apply-templates select="$chunk"/>
        </xsl:for-each-group>
    </xsl:copy>
</xsl:template>
```

I can use the presence of the subject name para, or the absence of a subhead, to discriminate the first chunk from the rest, like this:

```
<xsl:template match="chunk[w:p/w:pPr/w:pStyle/@w:val='entry-1name']"
priority="10">
    <xsl:apply-templates mode="subject" select="."/>
    <xsl:apply-templates mode="biography" select="."/>
</xsl:template>

<!-- the rest of the entry -->
<xsl:template match="chunk">
    <xsl:apply-templates mode="process"/>
</xsl:template>
```

For the subject and biography parts of the first chunk I need to do two different things. For the subject, I need to provide the <name>, and generate a sortable version, <sort-name>, from it. And I need to do that for any aliases the subject had too. Because we have a fixed output order in mind, and it's unambiguous what's a name or alias and what isn't, we can apply-templates with a mode and not worry about document order, filtering in the templates:

```
<xsl:mode name="process" on-no-match="shallow-skip"/>

<xsl:template match="chunk" mode="subject">
    <subject>
        <xsl:apply-templates select="*[w:pPr/w:pStyle/
@w:val='entry-1name']" mode="process"/>
```

```
        <xsl:where-populated>
            <aliases>
                <xsl:apply-templates select="*[w:pPr/w:pStyle/
@w:val='entry-0alias']" mode="process"/>
            </aliases>
        </xsl:where-populated>
    </subject>
</xsl:template>
```

Note the use of `on-no-match="shallow-skip"` on the `process` mode. Doing that lets us match whatever we're interested in without worrying about extra text-nodes littering our output, which is, as already mentioned, extremely sensitive to whitespace being added or removed. Handling the biography content is simply a case of outputting the right element and then ignoring the names when applying templates:

```
<xsl:template match="chunk" mode="biography">
    <biography>
        <xsl:apply-templates select="*[not(
                w:pPr/w:pStyle/@w:val = ('entry-1name', 'entry-0alias')
            )]" mode="process"/>
    </biography>
</xsl:template>
```

The biography section can contain multiple styles of paragraph: standard text, and lists, so we need to be able to group the flat list paras and wrap them into a list-container-with-list-items style lists. Grouping this kind of paragraph-level content relies on adjacency rather than a starting marker. `<xsl:for-each-group group-adjacent="…">` provides the means to achieve this. However, it's actually easier (in this case) to do that grouping at the outputting-HTML stage, so we'll return that to that for paragraphs later in the paper. We do need to figure out which paragraphs will eventually become list items, though. To do that we're going to use a trick where we store the mapping between input style name and desired element in a sequence of elements in a variable. We can use standard XPath pattern matching to get what we need, with minimal fuss:

```
<xsl:variable name="list-section-elements" as="element()+">
    <writings style-name="entry-4writings"/>
    <lit style-name="entry-5lit"/>
    <exhibitions style-name="entry-6exh"/>
    <collections style-name="entry-7coll"/>
</xsl:variable>

<xsl:template match="w:p[w:pPr/w:pStyle/@w:val = $list-section-elements/
@style-name]"
    mode="process" priority="10">
```

```
    <li><xsl:apply-templates select="." mode="para-guts"/></li>
</xsl:template>
```

If the <w:p> we're processing has a style name that matches the style-name attribute of one of our $list-section-elements, it's supposed to be a list item. We'll come back to processing the guts of paragraphs in a minute. Now we've handled matching paras in the first chunk, we can turn to the other chunks, which are effectively all containers for lists. All the bibliographic entries should occur together in the source, as should all the exhibitions, and so on. We can reuse our $list-section-elements tactic above, together with an XSLT 3 addition to <xsl:copy>:

```
<xsl:template match="chunk">
    <xsl:variable name="style-name" select="w:p[1]/w:pPr/w:pStyle/
@w:val"/>
    <xsl:copy select="$list-section-elements[@style-name = $style-name]">
        <xsl:apply-templates mode="process" select="$chunk"/>
    </xsl:copy>
</xsl:template>
```

The approach here is to select the element from $list-section-elements that matches the style name used by the paragraphs in the chunk, and copy it to wrap the chunk elements, which we can then process. select="…" on <xsl:copy> is an XSLT 3 feature, which makes this very simple, because we don't need to extract the element name of the wrapper element to feed to <xsl:element>.

Unfortunately, real life isn't as simple as that. This is a long manuscript and if there are errors, or some edge cases where the content model legitimately differs, we don't want to wind up with $style-name containing a multi-item sequence: offering multiple elements to <xsl:copy select="…"> will cause a dynamic error and halt processing.

To handle this case, we can get all the paragraph style names in the chunk, throw out ones not related to the section type (probably the legitimate edge cases), and then match that against our section elements. If we get more than one section element back, we can send an <xsl:message> and then just pick the first section element:

```
<xsl:template match="chunk">
    <xsl:param name="name" tunnel="yes"/>

    <xsl:variable name="style-name"
        select="distinct-values(w:p/w:pPr/w:pStyle/@w:val)
            [. = $list-section-elements/@style-name]"/>
    <xsl:variable name="chunk" select="."/>

    <xsl:if test="count($style-name) gt 1">
        <xsl:message expand-text="yes">Entry {$name} has multiple
```

```
section-type paras
            ({string-join($style-name, ', ')}) in a single chunk</
xsl:message>
    </xsl:if>
    <xsl:copy select="$list-section-elements[@style-name = $style-
name[1]]">
        <xsl:apply-templates mode="process" select="$chunk"/>
    </xsl:copy>
</xsl:template>
```

To facilitate sending a useful message, the original template that matches the <entry> generated by the splitting function adds the subject's name as a tunnelling param, which can then be used when needed, as we did above. It adds a couple of lines to the template:

```
<xsl:template match="entry">
    <xsl:variable name="name" select="string(w:p[w:pPr/w:pStyle/
@w:val='entry-1name'])"/>
    <xsl:copy>
        <xsl:for-each-group select="*" group-starting-with="w:p[w:pPr/
w:pStyle/@w:val='entry-2subhead']">
            <xsl:variable name="chunk">
                <chunk>
                    <xsl:sequence select="current-group()"/>
                </chunk>
            </xsl:variable>

            <xsl:apply-templates select="$chunk">
                <xsl:with-param name="name" tunnel="yes" select="$name"/>
            </xsl:apply-templates>
        </xsl:for-each-group>
    </xsl:copy>
</xsl:template>
```

All the section chunks are just lists with no other content, and we already have a template that will turn all the relevant <w:p>'s into 's, all we really need to do now is handle the contents of a paragraph (or list item). This requires us to deal with removing Word markers, concatenating runs of the same-styled text node, and generating the right output without altering the para text's white-space. It was surprisingly straightforward:

```
<xsl:template match="w:p" mode="process">
    <p><xsl:apply-templates select="." mode="para-guts"/></p>
</xsl:template>

<xsl:template match="w:p" mode="para-guts">
    <xsl:if test="w:pPr/w:pStyle/@w:val">
```

```
        <xsl:attribute name="class" select="w:pPr/w:pStyle/@w:val"/>
    </xsl:if>
    <xsl:for-each-group select="w:r" group-adjacent="if (self::w:r[w:rPr/
w:rStyle]) then self::w:r/w:rPr/w:rStyle/@w:val else '1'">
        <xsl:sequence select="if (current-grouping-key() = '1') then
gd:unstyled-runs(current-group()) else gd:styled-runs(current-group())"/>
    </xsl:for-each-group>
</xsl:template>

<xsl:function name="gd:unstyled-runs" as="text()?">
    <xsl:param name="runs" as="element()*"/>

    <xsl:value-of select="$runs/w:t" separator=""/>
</xsl:function>

<xsl:function name="gd:styled-runs">
    <xsl:param name="runs" as="element()*"/>

    <span class="{$runs[1]/w:rPr/w:rStyle/@w:val}">
        <xsl:value-of select="$runs/w:t" separator=""/>
    </span>
</xsl:function>
```

The first template is just the should-be-a-para equivalent of the should-be-a-list-item template we saw earlier. The mode="para-guts" template does the heavy lifting. Because WordprocessingML text elements inside a paragraph are as flat as the paragraphs are within the document, we can take advantage of the fact that all the Word non-text elements we want to throw away are only ever siblings of `<w:r>` elements. Through the magic of selecting just the `<w:r>` elements when we start our processing, we eliminate all the things we're uninterested in. Next we need to regroup all the runs of identically-styled text that were broken apart by the Word non-text elements we have just ignored out of existence. To do this, we turn to `<xsl:for-each-group group-adjacent="…">`. Unlike `group-starting-with`, the return value of the `group-adjacent` expression is used to generate a grouping key, and adjacent items in the input with the same grouping key are returned in the same group. The value returned by the expression must be a single item, not a multi-item sequence, and cannot be the empty sequence either, which is why we generate the value `"1"` as a placeholder for those text runs that do not have a character style applied. Once we have a group, it's a simple matter of either concatenating all the text nodes in the group into one, or doing that and also wrapping the result in a `<span class="{$style-name}">` element. The new-in-XSLT 3 `separator` attribute on `<xsl:value-of>` allows us to ensure that no extra whitespace is being added.

3.4. URL-slugs, the index, and writing all the entries to disk

Now we have a method for generating individual XML documents we need to think about how we output them to disk, and how we will generate the alphabetical index we need. Clearly, <xsl:result-document> is the mechanism we need to actually output the documents, but what should the filenames be? We're intending this to be a website, so the obvious thought is that the filenames should be valid and convenient parts of a URL path. We can define a transformation from the unicode-letters-and-spaces world of the subject name to the ASCII-only standard URL path, so that's not an especially hard problem. We could also, in theory, generate the index by crawling the generated entry document files. There is, however, a problem with both of these approaches, at least in the case of this project. The obvious title of these documents is the name of the subject, and the obvious URL slug is that name transformed into a valid path segment. However, the subject of these documents are people, and different people sometimes have the same name, which would generate the same URL slug.

That, in turn, raises the question of how you know if there's a duplicate name, and what to do about it. We are going to use a map/reduce approach using `<xsl:iterate>` to tackle all these problems. Firstly, knowing if there's a duplicate URL slug requires that we keep track of the URL slugs we've generated. Keeping track requires some sort of index, which we can later reuse as the source for the alphabetic index.

`<xsl:iterate>` was introduced for streaming processing, but provides the right kind of hooks to do the map/reduce job we need here. We can pass an initial value via a parameter to it, and at the end of each iteration set that parameter for the next iteration. If we pass in an empty map to the first iteration, and add each entry using its URL slug as the key as we go, we can use that map to check if a proposed new URL slug already exists, and modify it appropriately. At the end of the process we'll have an index containing every entry in the file:

```
<xsl:output method="json" indent="yes"/>
<xsl:output name="entry-xml" method="xml" indent="yes" suppress-
indentation="p li"/>

<xsl:function name="gd:as-entries" as="element(entry)*">
    <xsl:param name="body" as="element()"/>

    <xsl:for-each-group select="$body/*"
        group-starting-with="w:p[w:pPr/w:pStyle/@w:val='entry-1name']">
        <entry>
          <xsl:sequence select="current-group()"/>
        </entry>
    </xsl:for-each-group>
</xsl:function>
```

```
<xsl:template match="w:body">
    <xsl:iterate select="gd:as-entries(.)">
      <xsl:param name="index" select="map { }" as="map(xs:string,
map(*))"/>
      <xsl:on-completion select="$index"/>

      <xsl:variable name="entry" as="element(entry)">
          <xsl:apply-templates select="."/>
      </xsl:variable>

      <xsl:variable name="entry-with-slug" as="element(entry)">
          <xsl:apply-templates select="$entry" mode="add-url-slug">
            <xsl:with-param name="index" select="$index"/>
          </xsl:apply-templates>
      </xsl:variable>

      <xsl:variable name="entry-slug" select="$entry-with-slug/@url-
slug" as="xs:string"/>

      <xsl:result-document format="entry-xml" href="{resolve-
uri(concat($entry-slug, '.xml'), $output-path)}">
          <xsl:sequence select="$entry-with-slug"/>
      </xsl:result-document>

      <xsl:next-iteration>
          <xsl:with-param name="index" select="map:put($index, $entry-
slug, gd:index-entry($entry))"/>
      </xsl:next-iteration>
    </xsl:iterate>
</xsl:template>
```

We've already seen the body of the `gd:as-entries` function above. Declaring it as a function rather than a template makes it very easy to plug into the `select="…"` of the `<xsl:iterate>`. The first `<xsl:param>` sets up the index map, which begins as an empty map. Using `<xsl:on-completion>` means that the $index param will be returned at the end of the iteration, with the value that was calculated for it in the `<xsl:next-iteration>` block. The `<xsl:next-iteration>` block in question simple adds the index entry generated for the current entry to the existing `$index` map and passes that as the new value of `$index` for the next entry in the iteration.

In between we process the entry, using the approaches discussed in the previous section, and then we process that to generate a URL slug for the entry. Post-processing the entry means we've access to the finished entry, where we've already pulled out the main subject name and all the aliases. Once we have the

entry with a URL slug, we use `<xsl:result-document>` to write the complete entry to a file in the directory `$output-path` (a global stylesheet param), appending `.xml` to the URL slug for the filename. When the iteration is finished, the complete index map is serialized as JSON and written to disk by the process that started the transform.

The real complexity here is in the URL slug generation:

```
<xsl:mode name="add-url-slug" on-no-match="deep-copy"/>

<xsl:template match="entry" mode="add-url-slug">
    <xsl:param name="index" as="map(*)"/>

    <xsl:copy>
        <xsl:attribute name="url-slug"
            select="gd:entry-slug($index, subject/name)"/>
        <xsl:apply-templates mode="#current"/>
    </xsl:copy>
</xsl:template>

<xsl:function name="gd:entry-slug" as="xs:string">
    <xsl:param name="index" as="map(xs:string, map(*))"/>
    <xsl:param name="name-node" as="node()"/>

    <xsl:variable name="slug"
        select="translate(lower-case($name-node)), ' ', '_')"/>

    <xsl:sequence select="gd:unused-entry-slug($slug, 1, $index)"/>
</xsl:function>

<xsl:function name="gd:unused-entry-slug" as="xs:string">
    <xsl:param name="slug" as="xs:string"/>
    <xsl:param name="count" as="xs:integer"/>
    <xsl:param name="index" as="map(xs:string, map(*))"/>

    <xsl:variable name="test-slug" as="xs:string"
        select="if ($count > 1) then concat($slug, '-', $count) else
$slug"/>

    <xsl:sequence select="if (not(map:contains($index, $test-slug)))
        then $test-slug
        else gd:unused-entry-slug($slug, $count + 1, $index)"/>
</xsl:function>
```

The template just calls the function to generate the URL slug, then adding a `url-slug` attribute to the `<entry>` with the generated value. The current state of the index is passed as a param to the template, and on to the `gd:entry-slug()`

function, which performs the transform needed to create a URL slug from the subject's name (a process which is massively simplified in this example, since it's unrelated to the problem at hand). With the base URL slug generated, the `gd:unused-entry-slug()` checks the index to see if the URL slug already exists, and if it doesn't then the base slug is returned. If it does already exist, the count is incremented and the function calls itself with the base slug and the count. When the count is greater than 1, it's appended to the base slug before being checked against the index. This continues until an unused URL slug is found.

The last step of the index creation process is the creation of a new index entry, which is done by the `gd:index-entry()` function:

```
<xsl:function name="gd:subject-name" as="map(xs:string, xs:string)">
    <xsl:param name="subject" as="element()"/>

    <xsl:sequence select="map { 'name': string($subject/name), 'sort-
name': string($subject/sort-name) }"/>
</xsl:function>

<xsl:function name="gd:index-entry" as="map(xs:string, item())">
    <xsl:param name="entry" as="element(entry)"/>

    <xsl:variable name="aliases" as="map(xs:string, xs:string)*"
        select="for $alias in $entry/subject/aliases/alias return
gd:subject-name($alias)"/>

    <xsl:map>
        <xsl:sequence select="gd:subject-name($entry/subject)"/>
        <xsl:sequence select="if (empty($aliases)) then () else map
{ 'aliases': array { $aliases } }"/>
    </xsl:map>
</xsl:function>
```

The final JSON index looks like this:

```
{
  …,
  "mila_hoffmann_lederer": {
    "name": "Mila Hoffmann-Lederer",
    "sort-name": "Hoffmann-Lederer, Mila",
    "aliases": [ { "name":"Mila Hoffmannlederer", "sort-
name":"Hoffmannlederer, Mila" } ]
  },
  …
}
```

Combining all the index files generated each source file is trivial, and we do it in the program that runs the transforms. Once all the source files are transformed,

we have a directory full of entry XML documents, and a complete index of all the entries across all the source files.

4. Generating a website

Generating the HTML pages for entries themselves is relatively straightforward, but in addition to generating those, we're also generating the index from a JSON file, so we'll take a quick tour through the process.

4.1. Generating the A–Z index page

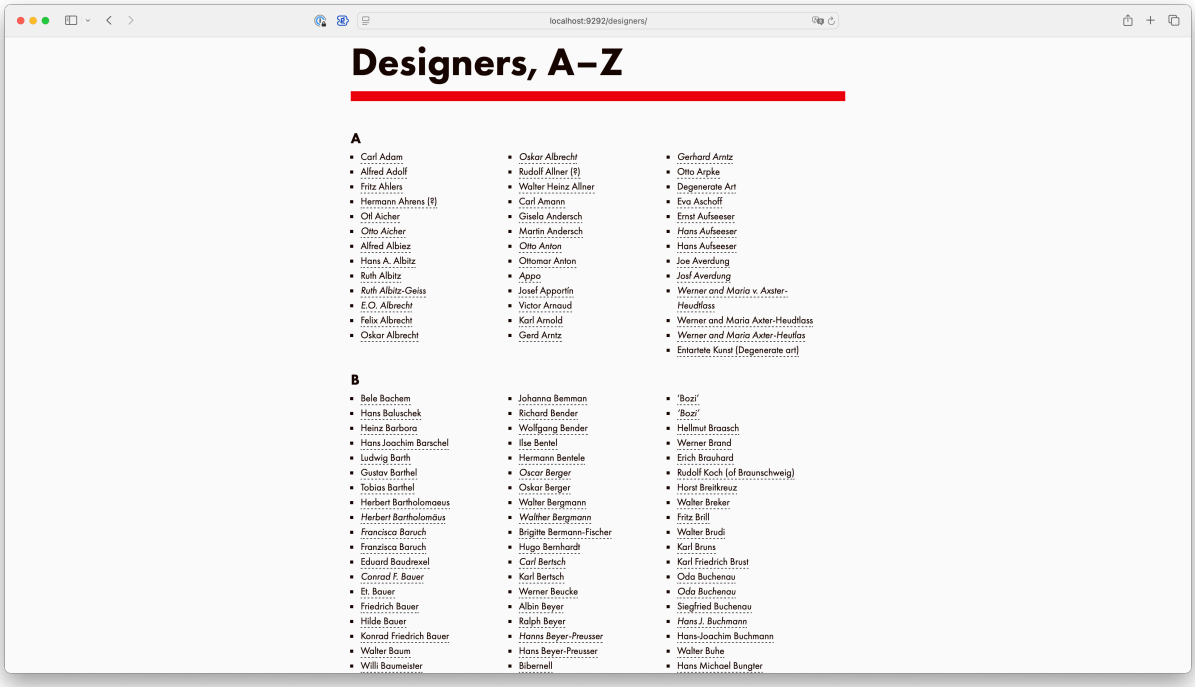


Figure 6. The A-Z index page

The index page is generated directly from the `index.json` file built from combining the indices built by processing the sources. To do this requires parsing the JSON file using a Saxon s9api.JsonBuilder, and then, in our case, calling a named template with the XdmMap just created as the global context item. We have to call a template rather than simply applying templates because we're importing a base layout stylesheet that controls the generation of the HTML skeleton, and a template that matches / won't match an XdmMap.

The layout stylesheet is very straightforward. It defines several modes, one for each 'hole' in the page that needs filling, generates the HTML skeleton and calls apply-templates in the right hole with right mode, which the importing stylesheets need to support. Here's how that looks for the index page, showing the areas for the masthead, content, and footer:

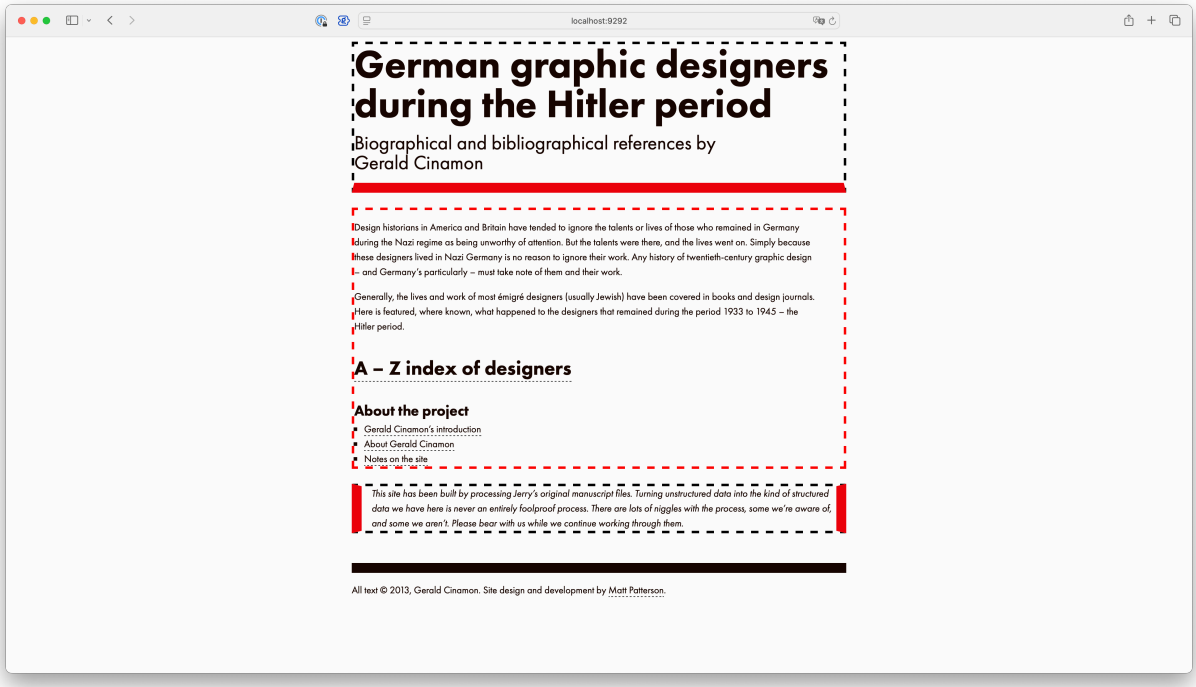


Figure 7. The layout skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  exclude-result-prefixes="xs math"
  version="3.0">

  <xsl:mode on-no-match="shallow-skip"/>
  <xsl:mode name="content" visibility="public" on-no-match="shallow-
skip"/>
  <xsl:mode name="masthead" visibility="public" on-no-match="shallow-
skip"/>
  <xsl:mode name="footer" visibility="public" on-no-match="shallow-
skip"/>

  <xsl:output method="html" html-version="5.0"/>

  <xsl:template name="index-template">
    <html>
      <head>
        <title>…</title>
        <link rel="stylesheet" media="all" href="/css/main.css" />
       </head>
      <body>
```

```
        <div id="container">
          <div class="masthead">
            <xsl:apply-templates mode="masthead" select="."/>
          </div>
          <div id="content">
            <xsl:apply-templates mode="content" select="."/>
          </div>
          <footer>
            <xsl:sequence>
              <xsl:apply-templates mode="footer" select="."/>
              <xsl:on-empty>
                <nav>
                  <div class="nav-group">
                    <h3><a href="/">Home</a></h3>
                  </div>
                  <div class="clearfix"/>
                </nav>
              </xsl:on-empty>
            </xsl:sequence>
            <div id="copyright">
              <p>…</p>
            </div>
          </footer>
        </div>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="/">
    <xsl:call-template name="index-template"/>
  </xsl:template>
</xsl:stylesheet>
```

Probably of most interest here is that we can provide navigation fallback through the use of `<xsl:on-empty>`, so if the footer templates generate nothing, a useful nav link back home is generated instead. The A-Z page stylesheet itself looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array"
  xmlns:gd="http://germandesigners.net"
  exclude-result-prefixes="xs math map array gd"
  version="3.0">
```

```
  <xsl:import href="../layouts/base.xsl"/>

  <xsl:template match=".[. instance of map(*)]" mode="masthead">
    <h1>Designers, A-Z</h1>
  </xsl:template>

  <xsl:template match=".[. instance of map(*)]" mode="content">
    <xsl:variable name="index" select="."/>

    <xsl:variable name="unordered-ungrouped-entries" as="element()*">
      <xsl:sequence select="map:for-each($index, gd:a-z-entries#2)"/>
    </xsl:variable>

    <xsl:for-each-group select="$unordered-ungrouped-entries"
        group-by="upper-case(substring(@sort-name, 1, 1))">
      <xsl:sort select="@sort-name"
          collation="http://www.w3.org/2013/collation/UCA?
strength=secondary"/>
      <h2 class="letter-index"><xsl:value-of select="current-grouping-
key()"/></h2>
      <xsl:variable name="max-col-entries" select="ceiling(count(current-
group()) div 3)"/>
      <xsl:for-each-group select="gd:sort-entries(current-group())"
        group-by="if (position() le $max-col-entries) then 1
            else if (position() le (2 * $max-col-entries)) then 2 else
3">
        <ul class="letter-index">
          <xsl:apply-templates select="current-group()"/>
        </ul>
      </xsl:for-each-group>
    </xsl:for-each-group>
  </xsl:template>

  <xsl:function name="gd:a-z-entries" as="element(entry)+">
    <xsl:param name="url-slug" as="xs:string"/>
    <xsl:param name="entry" as="map(*)"/>

    <entry url-slug="{$url-slug}" sort-name="{$entry?sort-name}"
name="{$entry?name}"/>
    <xsl:for-each select="array:flatten($entry?aliases)">
      <entry url-slug="{$url-slug}" sort-name="{.?sort-name}" name="{.?
name}" alias="true"/>
    </xsl:for-each>
  </xsl:function>
```

```
  <xsl:function name="gd:sort-entries" as="element()*">
    <xsl:param name="input" as="element()*"/>

    <xsl:perform-sort select="$input">
      <xsl:sort select="@sort-name" collation="http://www.w3.org/2013/
collation/UCA?strength=secondary"/>
    </xsl:perform-sort>
  </xsl:function>

  <xsl:template match="entry[@alias]">
    <li class="alias"><a href="/designers/{@url-slug}"><xsl:value-of
select="@name"/></a></li>
  </xsl:template>

  <xsl:template match="entry">
    <li><a href="/designers/{@url-slug}"><xsl:value-of
select="@name"/></a></li>
  </xsl:template>
</xsl:stylesheet>
```

The key stages are:

1. Output the page heading into the `masthead` hole.
2. To populate the `content` hole with the A–Z list, first generate a sequence of `<entry>` elements from `index.json`'s map, exploding out aliases so that each also has an `<entry>`.

   Because maps are unordered in XSLT 3, this sequence is unordered.
3. Use `<xsl:for-each-group>` to group alphabetically, by the first letter of the sort name. This gives us our A–Z buckets.
4. We want three columns of entries for each letter, so take the count of entries in a bucket, divide by 3 and round up to the next integer to get a maximum. Use `<xsl:for-each-group>` again, this time passing it a sorted sequence of entries, grouping into columns by checking if the position of the entry is <= the maximum (column 1), twice the maximum (column 2), or more (column 3).
5. Use `<xsl:apply-templates>` to generate the list items for each column, adding a different class to aliases

We use several XSLT 3-specific approaches here. Firstly, we use atomic-value template matching in order to allow the standard base layout to call apply-templates, selecting and matching on the map itself. In order to generate the exploded list of entries plus their aliases, we use `map:for-each()`, which calls the provided function for every key/value pair in the map, returning a single sequence containing the return values of the provided function. While you can use a anonymous function defined in line (`map:for-each($m, function($k,`

`$v) { … })`), we're using higher-order functions to pass the `gd:a-z-entries()` function in. The notation `gd:a-z-entries#2` says to select the two-argument form, in order to deal with choosing between function overloads. This approach allows us to effectively combine XSLT's standard element constructors with XPath expressions, which allows for readable, idiomatic, code.

4.2. A page for the entry

The entries themselves vary wildly in length and complexity. We've already done the majority of the work needed to generate the HTML during the creation of the entry XML, but there are a few things that need to be done: Wrapping list-items in the biography section in a `<ul>`, resolving and generating links for cross-references, and generating the previous / next links.

Here's an entry that contains very little. There's no biography, and there are no cross-references, so we're just left with the previous and next links.

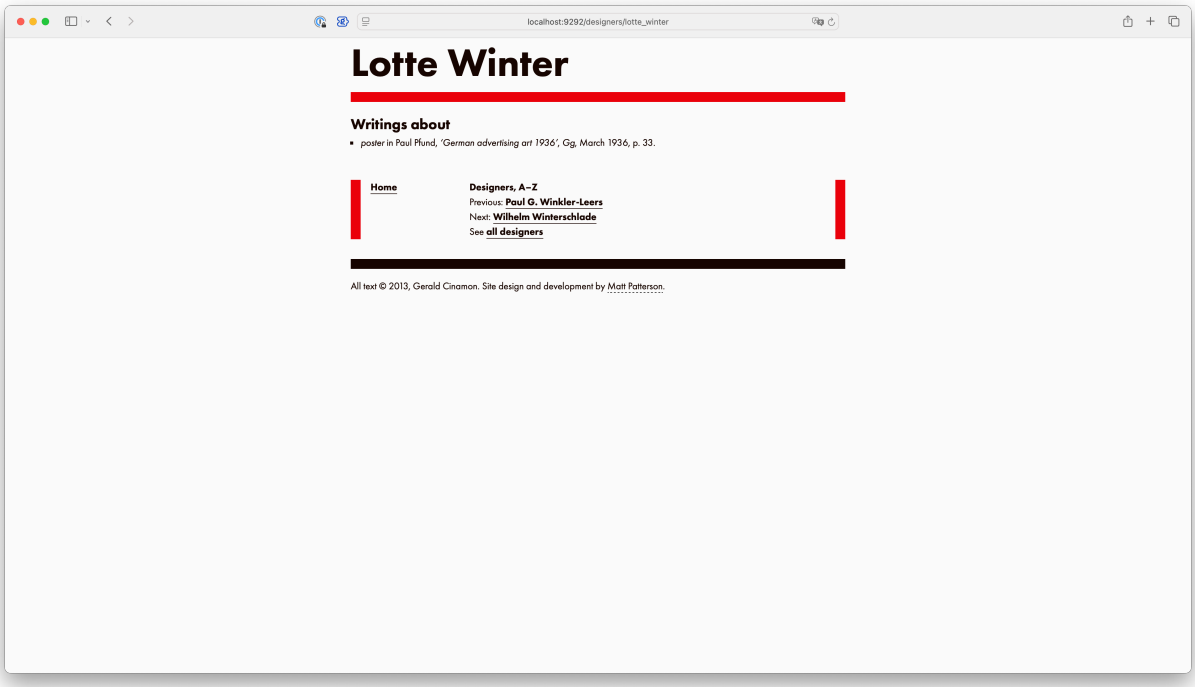


Figure 8. A short entry

To generate the previous and next links, we need to know the name and URL slug of both. While I'm sure we could do that with a query over a collection returned by `fn:collection()`, that seems a little wasteful: we'd need to perform it again for every entry. Instead, we're going to post-process our `index.json` and generate a new index that maps an entry (via its URL-slug) to the previous and next entries.

The XSLT to do this is quite compact, but demonstrates a couple of other nice XSLT 3 features, so I'll reproduce it here in full:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:math="http://www.w3.org/2005/xpath-functions/math"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:gd="http://germandesigners.net/"
  exclude-result-prefixes="xs math map"
  version="3.0">

  <xsl:output method="json" indent="yes"/>

  <xsl:function name="gd:sorted-index" as="array(*)*">
    <xsl:param name="index" as="map(*)"/>

    <xsl:sequence
      select="sort(map:for-each($index, function($key, $value) { [ $key,
$value ] }),
        'http://www.w3.org/2013/collation/UCA?strength=secondary',
        function($item) { $item(2)?sort-name })"/>
  </xsl:function>

  <xsl:function name="gd:sibling" as="map(xs:string, map(xs:string,
xs:string))">
    <xsl:param name="rel" as="xs:string"/>
    <xsl:param name="entry" as="array(*)"/>

    <xsl:sequence select="map { $rel: map {
          'url-slug': $entry(1), 'name': $entry(2)?name }
        }"/>
  </xsl:function>

  <xsl:variable name="gd:previous" select="gd:sibling('previous', ?)"
    as="function(array(*)) as map(*)" />
  <xsl:variable name="gd:next" select="gd:sibling('next', ?)"
    as="function(array(*)) as map(*)" />

  <xsl:template match=".[. instance of map(*)]" as="map(*)*">
    <xsl:map>
      <xsl:iterate select="gd:sorted-index(.)">
        <xsl:param name="previous" as="array(*)?" select="()"/>
        <xsl:on-completion select="map:entry($previous(1),
$previous(2))"/>

        <xsl:sequence select="if (empty($previous)) then ()
```

```
            else map:entry($previous(1), map:merge(($previous(2),
$gd:next(.))))"/>

        <xsl:next-iteration>
          <xsl:with-param name="previous"
            select="if (empty($previous)) then .
                else array { .(1), map:merge((.(2),
$gd:previous($previous))) }"/>
        </xsl:next-iteration>
      </xsl:iterate>
    </xsl:map>
  </xsl:template>
</xsl:stylesheet>
```

As you can see, we're using the same map/reduce through `<xsl:iterate>` technique from the original index-generation stylesheet. Again, we have to sort the input map because maps are unordered in XPath 3. We are using it in a different way, though. We need to add the previous and next references to every entry. The brute-force approach is to use `<xsl:for-each>` to iterate over the range `1 to count($sorted-entries)`, and subtract 1 from the current value to find the previous, and add 1 to find the next. You also need special cases for the first and last items, which won't have a previous or next item, respectively. The `<xsl:iterate>` approach here removes almost all the special casing and messing about. The approach relies on `<xsl:next-iteration>` and `<xsl:map>`.

`<xsl:map>` expects a sequence of maps as the result of evaluating its body, and it then merges them all into a single map. We'll return single-entry maps from the body of `<xsl:iterate>` so that we end up with the complete index map we need. Because maps are unordered, and we need to supply the entries correctly sorted (so that we can know the previous entry) we need to use an array or sequence. `map:for-each()` makes it trivial to output a sequence of key/value pairs, as a 2-item array. In addition to the current item in the sequence we pass the previous entry map as the `$previous` param, through the `<xsl:next-iteration>` mechanism. Now we can add the current entry to the previous as its `next` relation, and output from the body of the `<xsl:iterate>`. We then add the entry from `$previous` to the current one as its `previous` relation, and then pass that to the next iteration as the value of `$previous` through `<xsl:next-iteration>`. At the end of the iteration `<xsl:on-completion>` outputs the final entry, and we're done. Obviously, the value of `$previous` is `()` for the first entry, so that first step just doesn't output anything.

The function that generates the previous and next maps is `gd:sibling()`, but we don't use that directly. Instead we use partial function application to create new function objects, stored in the `$gd:previous` and `$gd:next` variables, that keep the invocation concise. Calling `$gd:next(.)` is the same as calling

`gd:sibling('next', .)`. We could, of course, write wrapper functions, but the end result would be the same and there would be more boilerplate. This case, with a simple single-variable binding is particularly suited because we don't need to process the variable being bound.

The `<xsl:map>` then creates a single map from all the output entries, and that gets serialized to disk to be used later.

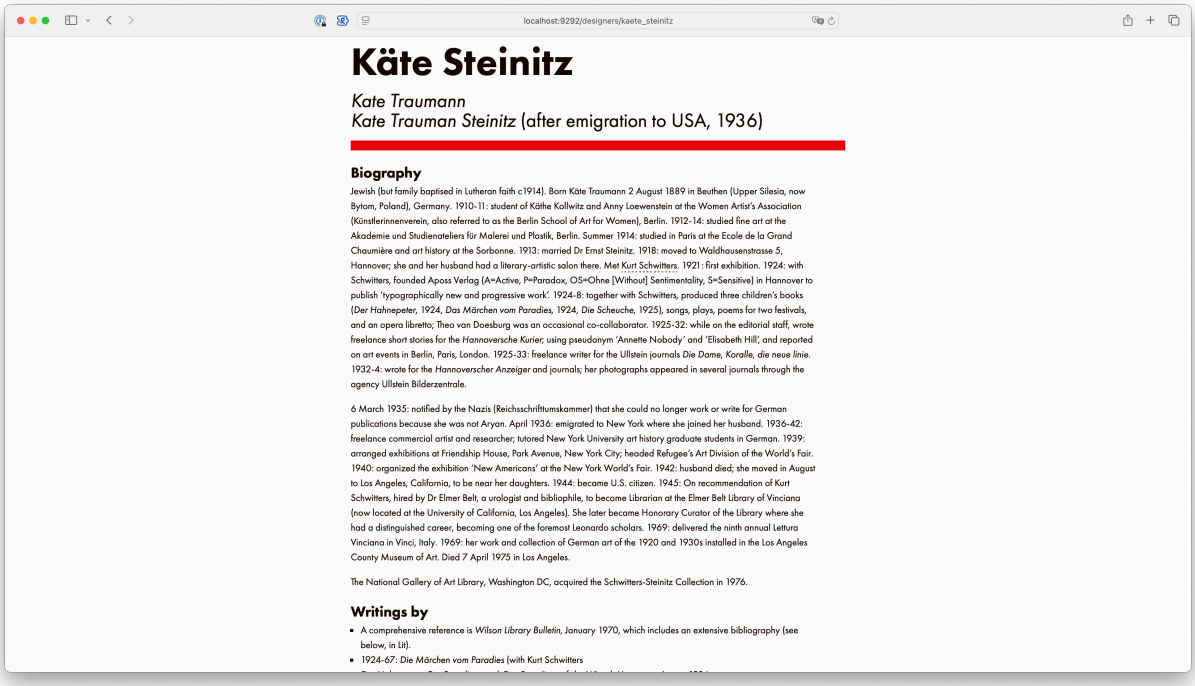


Figure 9. A long entry

We also need to look up cross-references. Again, while we could make this work with queries over an `fn:collection()`, what we're talking about is string comparison of the text content of a cross-reference `<span>` and the set of names we already have in the JSON index we produced for the A–Z page. We do that transformation in the build script, which is in Ruby [8]. That looks like this, where the collection (`index`) being operated on by `reduce` is passed an initial value, an empty map, which will be the left-most parameter in the block, `links`. The current item from `index` being evaluated is destructured to the `url_slug` and `entry` variables, added to the map in `links`, and the return value of the block is passed as the `links` parameter to the next iteration:

```
index.reduce({}) { |links, (url_slug, entry)|
  links[entry['name']] = url_slug
  if entry.has_key?('aliases')
    entry['aliases'].each { |aka|
      links[aka['name']] = url_slug
```

```
    }
  end
  links
}
```

Another map/reduce that winds up spitting out a very simple map with the name as key and URL-slug as value. With that in hand, let's start looking at the XSLT for generating the page, just including the bits relevant to the next/previous links and cross-refs for now:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:map="http://www.w3.org/2005/xpath-functions/map"
  xmlns:array="http://www.w3.org/2005/xpath-functions/array"
  exclude-result-prefixes="xs map array"
  version="3.0">

  <xsl:import href="../layouts/base.xsl"/>

  <xsl:param name="entry-links" as="map(*)" select="map {}"/>
  <xsl:param name="cross-refs" as="map(*)" select="map {}"/>

  <xsl:mode on-no-match="shallow-skip"/>
  <xsl:mode name="html" on-no-match="shallow-copy"/>

  <xsl:template match="span[@class = 'Crossref-designer'][$cross-
refs(string(.))]" mode="html">
    <span class="Crossref-designer"><a href="{ $cross-refs(string(.)) }">
      <xsl:apply-templates mode="#current"/>
    </a></span>
  </xsl:template>

  <xsl:template match="entry" mode="footer">
    <nav>
      <div class="nav-group">
        <h3><a href="/">Home</a></h3>
      </div>
      <div class="nav-group">
        <h3>Designers, A-Z</h3>
        <xsl:variable name="links" select="$entry-links(@url-slug)"/>
        <xsl:if test="map:contains($links, 'previous')">
          <p class="previous">Previous: <a href="/designers/{ $links?
previous?url-slug }">
            <xsl:value-of select="$links?previous?name"/></a>
          </p>
        </xsl:if>
```

```
        <xsl:if test="map:contains($links, 'next')">
          <p class="next">Next: <a href="/designers/{ $links?next?url-
slug }">
            <xsl:value-of select="$links?next?name"/></a>
          </p>
        </xsl:if>
        <p class="atoz">See <a href="/designers/">all designers</a></p>
      </div>
      <div class="clearfix"/>
    </nav>
  </xsl:template>
</xsl:stylesheet>
```

We pass the cross-ref and the next/previous index indices in as global params. They are, after all, the same dataset needed across all the entries we'll be processing. We can parse them once in the controlling program and reuse them in every transformation.

Cross-ref processing is very dumb: it just uses the text of the node as the string key into the $cross-refs map, and the template only matches if there's a match, so we don't even need to handle match and non-match behaviour, we just add a `<a href="…">` wrapper with the correct URL slug (they're relative links, so the URL-slug by itself is correct) and we're done. Some more processing needs to be done to the node text to make the lookup process more robust – currently, trailing whitespace would stop a lookup – but it demonstrates the approach.

The lookup needed for the next/previous links is much more robust – the URL-slug key is not pulled from a text node with dubious whitespace: it's exact, and unique. With the entry's links in hand, generating the HTML is a simple matter of adding `<p>`'s for the previous and next links, if they're there. Because the map keys are simple `xs:string`s whose value we already know, we can use the lookup operator `?` to pull them out: `$links?next?url-slug` being equivalent to `$links('next')('url-slug')` or `map:get(map:get($links, 'next'), 'url-slug')`.

Generating the Subject name in the masthead region is very straightforward:

```
<xsl:template match="name[parent::subject]" mode="masthead">
    <h1><xsl:apply-templates mode="html"/></h1>
</xsl:template>

<xsl:template match="aliases" mode="masthead">
    <ul class="aka">
        <xsl:apply-templates mode="#current"/>
    </ul>
</xsl:template>

<xsl:mode name="alias" on-no-match="shallow-skip"/>
```

```
<xsl:template match="alias" mode="masthead">
    <li><xsl:apply-templates mode="alias"/></li>
</xsl:template>

<xsl:template match="name" mode="alias">
    <xsl:apply-templates mode="html"/>
</xsl:template>
```

The most complex part is handling any aliases, because we need to wrap that in a list, but because of the `<aliases>` container there's a very obvious place to hang that.

Dealing with the various sections is essentially the same thing repeated several times with different headings, so we'll just look at the biography section, before looking at the `html` mode:

```
<xsl:template match="biography" mode="content">
    <xsl:call-template name="section">
        <xsl:with-param name="class" select="'description'"/>
        <xsl:with-param name="heading">Biography</xsl:with-param>
    </xsl:call-template>
</xsl:template>

<xsl:template name="section" expand-text="yes">
    <xsl:param name="heading" as="xs:string"/>
    <xsl:param name="class" as="xs:string"/>

    <div class="{$class}">
        <h2>{$heading}</h2>
        <xsl:for-each-group select="*"
            group-adjacent="if (self::li) then self::li/@class else
false()">
            <xsl:choose>
                <xsl:when test="current-grouping-key()">
                    <ul>
                      <xsl:apply-templates select="current-group()"
mode="html"/>
                    </ul>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:apply-templates select="current-group()"
mode="html"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each-group>
    </div>
</xsl:template>
```

```
<xsl:template match="p|li" mode="html">
    <xsl:copy>
        <xsl:apply-templates mode="html"/>
    </xsl:copy>
</xsl:template>
```

Because the biography section con contain lists as well as standard paras, we make use of `<xsl:for-each-group group-adjacent="…">` for the last time. If the element is an `<li>`, then we use its `@class` for the grouping key, otherwise just return `false()`, which is a value we can guarantee that none of the `xs:string` class attribute values will be, and which allows us to use the delightfully simple `<xsl:when test="current-grouping-key()">` to switch between list-wrapping and normal processing.

Normal processing is also very simple, by design. In the entry XML generation phase we created content which used HTML elements, so all we need to do, unless we have a special case, is copy the nodes into the output. The special cases are cross-refs, which we've covered already, and stripping extra attributes from the `<p>` and `<li>` elements – the class attributes we pulled over from Word but no longer need now we've wrapped list items. We use `<xsl:copy>` to copy the element, but not its attributes, and go on our way - the `html` mode is `on-no-match="shallow-copy"`, so everything we don't explicitly process just gets copied over.

5. In conclusion

This paper aimed to stand as a useful behind-the-scenes exploration of all the stages of preparation and processing the German Designers project required, explaining the choices we made and showing the techniques we used, in the hope that they are useful to people.

After a number of years of stable deployment without updates, the need to move from the original dynamic website, the desire to move to statically generated pages, and the existing XSLT 2 WordprocessingML transforms, gave us the motivation, and an ideally sized playground, to explore the possibilities XSLT and XPath 3 far more thoroughly in a real-world setting than we expected. The results, in terms of the relative simplicity of the XSLT and build system code, and its speed of development, was delightful. An area like JSON processing, which has not been traditionally regarded as a strong point of XSLT / XPath was easy to integrate and allowed not only easy sharing of conveniently-queryable data structures between processing stages, but also easy interoperability with non-XSLT tooling.

We look forward to being able to explore some of the initial design ideas for the project that previously seemed too complex, now that we have tooling that fits the problem space, and tooling that permits such easy iterative development.

6. Bibliography

Bibliography

[1] Gerald Cinamon: *Gerald Cinamon, Graphic Designer and Author*. https://geraldcinamon.com/

[2] ICA, London: *Gerald Cinamon: Collected Work Since 1958*. https://archive.ica.art/whats-on/gerald-cinamon-collected-work-1958/

[3] Gerald Cinamon: *German graphic designers during the Hitler period*. http://www.germandesigners.net/

[4] Evan Lenz: *The WordprocessingML Vocabulary*. https://lenzconsulting.com/wordml/

[5] Wikipedia: *Office Open XML file formats*. https://en.wikipedia.org/wiki/Office_Open_XML

[6] ISO: *Office Open XML File Formats — Part 1: Fundamentals and Markup Language Reference*. https://www.iso.org/standard/71691.html

[7] ECMA International: *ECMA-376*. http://ecma-international.org/publications-and-standards/standards/ecma-376/

[8] Ruby contributors: *The Ruby programming language*. https://www.ruby-lang.org/

[9] Ruby on Rails contributors: *Ruby on Rails*. https://rubyonrails.org/

Enabling AI Across the XML Technologies via XPath Functions

George Bina
<george@oxygenxml.com>

Abstract

XPath is the shared expression language for the XML technology family. XSLT, XQuery, Schematron, SQF, XProc, and XSpec all evaluate XPath expressions. This makes XPath the natural integration point for adding AI capabilities to the entire XML toolchain.

This paper describes two generations of AI XPath functions. The first generation - `ai:transform-content`, `ai:verify-content`, and `ai:invoke-action` - was implemented as Saxon extension functions in Oxygen AI Positron to allow experimenting with AI from XML technologies. These functions enabled AI-powered content generation, semantic validation, and automatic fixes across XSLT stylesheets, XQuery scripts, Schematron schemas, and XProc pipelines. The paper shows how they were used in practice and discusses their technical and portability limitations.

The second generation centers on a new function, `ai:chat`, which returns the full conversation history in addition to the AI response. This makes multi-turn workflows and richer pipelines possible. Beyond the technical improvements, this paper proposes `ai:chat` as a community standard, following the EXPath model, so that AI-enabled XML applications are portable across processors and independent of any specific AI provider or tool.

Keywords: AI, XPath, XSLT, XQuery, Schematron, XProc, ai:chat, EXPath, XML

1. Introduction

XML technologies are great at structure, precision, and deterministic transformation and validation. AI models are great at meaning, interpretation, and reasoning about natural language. Connecting the two opens up a class of applications that neither can achieve alone - generating content that is both structurally correct and semantically coherent, validating documents against rules that require natural-language understanding, and building publishing workflows that adapt intelligently to content.

There are two ways to integrate AI with the XML toolchain. The first is AI using XML tools, where an AI agent orchestrates a task and calls XSLT transfor-

mations, Schematron validators, or XProc pipelines as tools in its reasoning loop. The second is the reverse - XML technologies using AI, where an XSLT stylesheet, a Schematron rule, or an XProc pipeline calls an AI model as one step in a deterministic workflow. This paper focuses on the second direction.

The natural integration point for this is XPath. Because XSLT, XQuery, Schematron, SQF, XProc, and XSpec all evaluate XPath expressions, a single XPath extension function makes AI available everywhere in the toolchain simultaneously, with no adapters, no glue code, no framework-specific APIs. An XSLT template can call an AI function the same way it calls `string-length()` or `tokenize()`.

This paper reports on a few years of experience building and using AI XPath functions in Oxygen AI Positron, describes the evolution from first-generation to second-generation function design, and argues that the community should standardize a portable `ai:chat` function in the same way EXPath standardized file I/O, HTTP access, and ZIP handling.

2. First Generation: AI XPath Functions in Oxygen AI Positron

Oxygen AI Positron introduced three AI XPath extension functions, implemented on top of the Saxon XSLT/XQuery processor. They are available wherever Oxygen evaluates Saxon XPath, which covers the full range of XML technologies supported by the product.

2.1. ai:transform-content

`ai:transform-content` sends content to an AI model with an instruction and returns the model's response as a plain string. The signature is:

```
ai:transform-content(
  $instruction as xs:string,
  $content     as xs:string,
  ($assistant  as xs:string,
   $user       as xs:string)*
) as xs:string
```

The `$instruction` argument becomes the system message, setting the AI's task and behaviour. The `$content` argument becomes the user message with the content to process. The optional part accepts zero or more assistant/user pairs that inject prior conversation turns, so the function can continue a dialogue when needed.

2.2. ai:verify-content

`ai:verify-content` asks the AI a yes/no question about content and returns `xs:boolean` directly, without requiring the caller to compare a string to "yes":

```
ai:verify-content(
  $question   as xs:string,
  $content    as xs:string
) as xs:boolean
```

The boolean return type is what makes this useful - the function can be used directly in a Schematron `test` attribute, an XSpec `select` expectation, or an XSLT predicate, without any string post-processing.

2.3. ai:invoke-action

`ai:invoke-action` invokes a named action defined in Oxygen AI Positron. The action definition stores the system prompt and many other parameters that control functions, model, model parameters, etc. keeping them out of the XSLT or Schematron source:

```
ai:invoke-action(
  $actionId    as xs:string,
  $extraPrompt as xs:string,
  $content     as xs:string
) as xs:string
```

`ai:invoke-action` has three arguments. The `$extraPrompt` argument lets callers append runtime context to the action's stored prompt - pass an empty string when nothing extra is needed.

3. AI Across the XML Toolchain

Because the functions are Saxon XPath extensions, they work wherever Oxygen evaluates Saxon XPath. The following sections show how each technology in the XML toolchain uses them.

3.1. XSLT

The most direct use is in XSLT templates. The following template adds a generated `shortdesc` element to any DITA topic that is missing one, using the full topic text as context for the AI:

Example 1. XSLT template generating a missing DITA short description

```
<xsl:template match="topic[not(shortdesc)]/title">
  <xsl:copy-of select="."/>
  <shortdesc>
    <xsl:value-of select="
      ai:transform-content(
        'Generate a short description in under 30 words:', ..)"/>
```

```
    </shortdesc>
  </xsl:template>
```

The double-dot `..` passes the full parent topic node as context, giving the model access to the title, body, and all child elements. The same pattern runs as an Oxygen XML Refactoring operation, processing an entire document set in a single pass.

3.2. XQuery

Because the functions are XPath 3.1 extensions, they work identically in XQuery. The following fragment uses XQuery Update Facility to insert an AI-generated short description into a topic that lacks one:

Example 2. XQuery Update fragment inserting a generated short description

```
if (not(shortdesc)) then
  insert node
    <shortdesc>{
      ai:transform-content(
        'Generate a short description in under 30 words:',
        .)
    }</shortdesc>
  after title
else ()
```

The call is identical to the XSLT version. The function is XPath, not XSLT-specific, so no adaptation is needed.

3.3. Schematron and Schematron Quick Fixes

Schematron's `test` attribute takes a boolean XPath expression. `ai:verify-content` returns `xs:boolean`, so it fits directly as a test condition and can catch things no structural pattern could. Combined with `ai:transform-content` in a quick fix, the same rule that detects a problem can also fix it:

Example 3. Schematron rule using AI for semantic validation and AI-powered quick fix

```
<sch:rule context="shortdesc">
  <sch:let name="shortdesc" value="string(.)"/>
  <sch:let name="body"      value="string(../body)"/>
  <sch:report
    test="ai:verify-content(
      'Is the shortdesc vague, generic, or off-topic for the topic
body?',
```

```
      'Shortdesc: ' || $shortdesc || ' Body: ' || $body)"
    role="warn" sqf:fix="rephrase">
    Short description may be vague or off-topic.
  </sch:report>
  <sqf:fix id="rephrase">
    <sqf:description>
      <sqf:title>Rephrase short description</sqf:title>
    </sqf:description>
    <sqf:replace match="text()"
      select="ai:transform-content(
        'Rephrase the shortdesc as a clear, focused description under 40
words:',
        'Shortdesc: ' || $shortdesc || ' Body: ' || $body)"/>
  </sqf:fix>
</sch:rule>
```

This single rule demonstrates both roles: the AI performs the check (detecting semantic problems that no structural XPath expression could catch), and the SQF quick fix generates the corrected text. The author clicks a button to apply the fix.

3.4. XProc

In XProc, an AI step can be built as a custom step type using an inline XSLT that calls the AI XPath functions. The following example from a real pipeline generates press releases from product release notes in two AI passes per item:

Example 4. XProc pipeline using custom AI steps for press release generation

```
<p:viewport match="//releaseNotes">
  <!-- Pass 1: draft the press release -->
  <oxygen:invokeAI name="generateDraft">
    <p:with-option name="promptref"
        select="resolve-uri('prompts/pressRelease.md')"/>
  </oxygen:invokeAI>
  <!-- Pass 2: adapt to company style -->
  <oxygen:invokeAI name="adaptStyle">
    <p:with-option name="promptref"
        select="resolve-uri('prompts/adaptToStyle.md')"/>
  </oxygen:invokeAI>
</p:viewport>
```

The `oxygen:invokeAI` custom step embeds an inline XSLT that calls the AI functions, reading the system prompt from a Markdown file. The `p:viewport` iterates over each release notes element, running both AI calls for each item. Prompts are kept in separate files outside the pipeline, which simplifies maintenance and prompt iteration.

3.5. Other Integration Points

Because the functions are implemented as Saxon extension functions, any Oxygen plugin or feature that uses Saxon XPath evaluation can call them. In practice this includes new file templates (where `${xpath_eval(...)}` evaluates XPath at file creation time), Oxygen XML Refactoring operations (both XSLT and XQuery variants), the Oxygen Git Client plugin (which uses them to generate commit message suggestions from staged file diffs), and the Oxygen CSS engine used for Author mode and Chemistry PDF publishing.

4. Testing AI Prompts

Prompt engineering without testing does not scale. As prompts evolve, we need to detect regressions, compare the effect of prompt changes, and build a repeatable baseline. The AI XPath functions support two testing approaches that can be combined: exact structural checks and semantic similarity checks.

4.1. XSpec

XSpec is the unit-testing framework for XSLT, XQuery, and Schematron. Because `ai:transform-content` and `ai:verify-content` are XPath functions, they can be called directly inside XSpec scenarios. The following scenario tests a short description generation function:

Example 5. XSpec scenario testing AI-generated content with structural and semantic assertions

```
<x:scenario label="Generate short description">
  <x:variable name="src" select="unparsed-text(
      'resources/apples.dita', 'UTF-8')"/>

  <x:call function="ai:transform-content">
    <x:param>Generate a DITA shortdesc element
for the following topic:</x:param>
    <x:param select="$src"/>
  </x:call>

  <x:expect label="Returns a shortdesc element"
    test="parse-xml($x:result)/*/name()"
    select="'shortdesc'"/>

  <x:expect label="Describes the source"
    test="ai:verify-content(
      'Is this a valid short description of the source?',
      $src || '&#10;Shortdesc: ' || $x:result)"
```

```
    select="true()"/>
</x:scenario>
```

The first assertion is structural: it parses the result as XML and checks that the root element is named `shortdesc`. The second assertion is semantic: it asks the AI whether the generated description is actually valid for the source topic. Both assertions run as standard XSpec expectations. The `$src` variable is declared once and shared between the function call and the verification step, ensuring the semantic check has access to the original content.

4.2. Schematron Test Cases as XML Data

An alternative approach stores test cases as XML data and drives them with a single Schematron rule. The test data file holds the prompt and source reference as attributes, with child elements for the expected structural and semantic outcomes:

Example 6. XML test data file

```
<tests>
  <test prompt="Generate a DITA shortdesc element for the following:"
        source="resources/apples.dita">
    <expectedRootElement>shortdesc</expectedRootElement>
    <expectedSimilarResult>Apples are [...]</expectedSimilarResult>
  </test>
  <!-- add more test elements here -->
</tests>
```

A single Schematron rule then iterates over every `test` element, calls the AI, and applies both a structural and a semantic assertion:

Example 7. Schematron rule evaluating AI output against XML-encoded test cases

```
<sch:rule context="/tests/test">
  <sch:let name="src"
    value="unparsed-text(resolve-uri(@source, base-uri(.)))"/>
  <sch:let name="result" value="ai:transform-content(@prompt, $src)"/>
  <sch:assert test="matches($result, '^<' || expectedRootElement)">
    Wrong root element.
  </sch:assert>
  <sch:assert test="ai:verify-content(
      'Is this similar in meaning to: ' || expectedSimilarResult,
$result)">
    Result not semantically similar.
  </sch:assert>
</sch:rule>
```

Adding a new test case requires only a new `test` element in the data file - the rule does not change. Schematron validation tooling runs all test cases in a single batch pass.

5. Limitations

Experience with the first-generation functions revealed two categories of limitations.

Technical limitations. The `ai:transform-content` function offer no per-call control over AI parameters: there is no way to specify the model, temperature, available tools, or any other backend option at the call site. Every call uses whatever configuration Oxygen AI Positron has globally set. The history continuation mechanism (the optional interleaved `$assistant`/`$user` string pairs) is difficult to use in practice and makes dynamic multi-turn workflows difficult to implement. The `ai:invoke-action` function is additionally constrained to exactly three arguments and supports no continuation at all. It also requires the action to be identified by an ID that Oxygen AI Positron has already loaded, so the action cannot be constructed dynamically at call time. The return value is always a plain string, with no access to error information, warnings, etc.

Portability limitation. The functions are Saxon extension functions registered by the Oxygen AI Positron plugin. An XSLT stylesheet or Schematron schema that calls `ai:transform-content` only runs inside Oxygen. The community has no shared namespace, no agreed function signatures, and no specification that a second implementer could follow. This prevents sharing AI-enabled XML applications across tools and prevents the community from building on a common foundation.

6. A Proposal for the XML Community: ai:chat

The XML community can agree on a portable extension function specification for AI: define a namespace, a function signature, a message format, and a set of standard option keys. That may be an EXPath module.

6.1. Design Goals

Three goals shaped the design:

- **Provider independence.** The function does not expose provider-specific parameters in the type signature. Provider selection and authentication are environment configuration, not author code. An XSLT stylesheet that calls `ai:chat` should run unchanged whether the backend is Claude, GPT-4o, Gemini, or a locally hosted model.

- **Composability.** The `?messages` array returned by `ai:chat` can be passed directly as part of the next call's `$messages` sequence, so multi-turn conversations need no external state management.
- **Minimal surface area.** A single function covers all common AI integration patterns. Authors who only need a single-turn string result can call `ai:chat('prompt')?response`, while authors building multi-turn pipelines use a full map as the `$messages` parameter.

6.2. The ai:chat Function

The proposed function signature is:

```
ai:chat($messages as item()*, $opts as map(*)?) as map(*)
```

`$messages` must contain at least one item - an empty sequence is rejected as a backend error. `$opts` is genuinely optional and can be omitted entirely. The namespace URI is `http://expath.org/ns/ai`, following the EXPath convention.

6.3. Message Format

The `$messages` parameter accepts a heterogeneous sequence of message items. Three item types are supported:

- `xs:string` — converted to a `user`-role message. This is the simplest case: `ai:chat('What is a DITA shortdesc?')`.
- `map(*)` with at least a `role` key and a `content` key — sent as-is. Known role values are `system`, `user`, `assistant`, and `tool`. Unknown role values are passed through to the backend unchanged, ensuring forward compatibility with future role additions.
- `array(*)` whose members are message maps — auto-expanded into individual items. This allows the `?messages` array from a prior `ai:chat` call to be spliced directly into the next call's sequence without any unwrapping.

The following example shows a multi-turn conversation where the history from the first call feeds directly into the second:

Example 8. Multi-turn conversation using ?messages auto-expansion

```
let $r1 := ai:chat('What is a DITA shortdesc?')
let $r2 := ai:chat(($r1?messages, 'Give me a concrete example.'))
return $r2?response
```

The `($r1?messages, 'Give me a concrete example.')` sequence mixes the prior history array (auto-expanded) with a new plain string. No explicit message wrapping is needed.

6.4. Options

The optional `$opts` map accepts the following keys, all optional:

Key	Type	Description
`model`	`xs:string`	AI engine/model identifier, overrides the connector default
`temperature`	`xs:double`	Sampling temperature, 0–2
`max-tokens`	`xs:integer`	Maximum output tokens
`on-error`	`xs:string`	`'throw'` (default) raises a dynamic error on failure; `'return'` returns the error in `?error` instead
`cache`	`xs:boolean`	When `true()` (default), the implementation may return a cached result for an identical prior call
`tools`	`xs:string`	JSON array of tool definitions available to the model

6.5. Return Value

The function returns a `map(*)` with the following keys:

- `?response` (`xs:string?`) — the final assistant reply as a plain string. Absent when `on-error='return'` and the call failed.
- `?messages` (`array(map(*))`) — the complete conversation history after the call: all input messages plus the assistant reply. Each member has at minimum `role` and `content` keys. This array may be passed directly as part of the `$messages` sequence of a subsequent `ai:chat` call to continue the conversation.
- `?error` (`xs:string?`) — present only when `on-error='return'` and a failure occurred. Contains a human-readable description of the failure.
- `?warnings` (`array(xs:string)`) — non-fatal diagnostic messages. Absent when empty.

The main difference from the first generation is the `?messages` return key: it gives the caller the full conversation history, so continuing a conversation is just passing that array into the next call.

6.6. Tool and Function Calling

Modern AI models support tool calling (also known as function calling): the model can request that specific functions be executed during a conversation turn, receive their results, and continue reasoning from there. The `tools` key in `$opts` accepts a JSON string containing an array of tool definitions. Each tool definition follows the standard structure used by major AI providers:

Example 9. Tool definition JSON passed via $opts?tools

```
[{
  "type": "function",
  "function": {
    "name": "get_document_content",
    "description": "Read content from a file or URL.",
    "parameters": {
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "type": "object",
      "properties": {
        "docURL":     { "type": "string" },
        "start_line": { "type": "integer" },
        "end_line":   { "type": "integer" }
      },
      "required": ["docURL", "start_line", "end_line"]
    }
  }
}]
```

The tool execution loop is handled transparently by the implementation: when the model requests a tool call, the implementation executes it, appends the result as a `tool`-role message, and re-submits the conversation. This loop continues until the model produces a final text response. From the XPath author's perspective, a call with tools is still a single `ai:chat` invocation that returns a `?response` string. The full trace, including all tool requests and results, is preserved in the `?messages` array.

Tool calling bridges the gap between the first integration direction described in the introduction (AI using XML tools) and the second (XML technologies using AI). When an `ai:chat` call has access to tools such as `search_project_resources`, `validate_content`, or `get_document_content`, it can act as a fully agentic step inside an XSLT pipeline or XProc workflow, reasoning over a document set, validating intermediate results, and deciding what to do next.

6.7. ai:chat Across the XML Toolchain

The function can replace and extend everything the first-generation functions did. The following examples show the same patterns rewritten with `ai:chat`:

Example 10. Single-turn short description generation

```
ai:chat((
  map{'role':'system', 'content':'You are a DITA expert.'},
  'Generate a shortdesc for: ' || .
))?response
```

Example 11. Semantic validation in a Schematron test attribute

```
ai:chat(
  map{'role':'system', 'content':'Reply only with yes or no'},
  'Is the shortdesc vague, generic, or off-topic for the topic body?',
  'Shortdesc: ' || $shortdesc || ' Body: ' || $body)"
)?response = 'yes'
```

Example 12. Multi-turn pipeline: summarise, then rewrite as short description

```
let $r1 := ai:chat((
  map{'role':'system', 'content':'You are a DITA expert.'},
  'Summarise this topic in 3 sentences: ' || .
))
let $r2 := ai:chat((
  $r1?messages,
  'Now rewrite that as a single DITA shortdesc element.'
))
return $r2?response
```

The multi-turn pattern is clean: `$r1?messages` is an `array(*)` that auto-expands when placed in a sequence, so the second call sees the full context of the first without any explicit serialisation.

7. Toward a Portable Standard

The proposal is for `ai:chat` to become a portable community standard, not just an Oxygen feature.

If Saxon, BaseX, oXygen, and other XPath processors implement the `http://expath.org/ns/ai` namespace with conforming `ai:chat` implementations, then:

- An XSLT stylesheet that calls `ai:chat` becomes tool-independent. The author does not need to know which AI backend is configured behind the function.
- A Schematron schema that uses `ai:chat` for semantic validation can be submitted to a standards body, published as open source, or shared between organisations using different XML tooling.

- The XML community gains a shared foundation for AI integration, rather than each tool vendor implementing incompatible proprietary functions.

A draft specification has been prepared in the xmlspec format, defining the namespace, message format, function signature, option keys, return map keys, and error codes. The draft is intended as a starting point for community discussion.

8. Conclusions and Future Work

XPath is already the integration point for the XML technology family. Making AI available as an XPath function requires no adapters, no framework-specific APIs, and no changes to existing tooling infrastructure. The first-generation functions proved the pattern works across XSLT, XQuery, Schematron, SQF, XProc, and XSpec. The second-generation `ai:chat` function addresses their technical limitations by returning conversation history, adding per-call model control, and supporting tool calling for agentic workflows.

The larger goal is standardisation. The XML community built EXPath to give portable functions across XPath processors. We propose to do the same for AI chat completion. A stable namespace, a minimal function signature, and an agreed set of option keys are enough to let multiple processors implement the same API and let authors write AI-enabled XML applications that are independent of any specific tool or AI provider.

As a next step, we are making the draft specification available, collecting feedback on the message format and option key set and providing a prototype implementation. A few open questions are worth community discussion. Should the specification support explicit client-side tool execution, where the caller handles each tool call rather than relying on transparent looping inside the implementation? Should a streaming mode, returning response tokens incrementally as they arrive, be in scope for `ai:chat` or addressed by a companion function? And should a structured output option, constraining the model to reply with a valid JSON object matching a given schema, be standardised as part of the `$opts` map?

Bibliography

[1] http://expath.org/spec/http-client. *HTTP Client Module — EXPath Candidate Module*. EXPath.

[2] https://www.w3.org/TR/xpath-31/. *XML Path Language (XPath) 3.1 — W3C Recommendation*. W3C.

[3] https://www.w3.org/TR/xslt-30/. *XSL Transformations (XSLT) Version 3.0 — W3C Recommendation*. W3C.

[4] https://www.iso.org/standard/85625.html. *ISO/IEC 19757-3:2025 — Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation using Schematron*. ISO.

[5] https://spec.xproc.org/master/head/xproc/. *XProc 3.1: An XML Pipeline Language*. W3C Community Group.

[6] https://schematron-quickfix.github.io/sqf/. *Schematron Quick Fixes (SQF)*. W3C Quick-fix support for XML Community Group.

[7] https://xspec.io/. *XSpec — BDD Framework for XSLT, XQuery and Schematron*.

Fento, an adjusted approach for Xml/Java object binding

Jorge Sanchez Rodriguez
Vionta.net
<jorgesanchez@vionta.net>

Abstract

Java XML binding tools tend to look for the same data structures and approach on the object and document side. They also tend to use Xml as a mere tool for object serialization.

We share a proposal to enhance the document side of Java / Xml binding. The proposal relies on a more suitable tool for referencing document elements (modern XPath), a non volatile approach on document contents, focusing on the meaningful parts of the information while avoiding unnecessary mappings on the object side. We will walk through examples to show the approach use cases, virtues, and also its limitations.

Keywords: XML, Java, Binding, XPath, Security, data complexity, data flexibility

1. Introduction

This document a proposal for an adjusted approach for object/Java document data binding. The proposal is backed by a sample implementation, Fento.

The approach main characteristics are:

- Defined by Java annotations, in a similar way to Java JAX-B and many other current Java configuration tools.
- The document information is preserved, only the significant parts need to be extracted and optionally updated.
- Object and document structure is not expected to share the same pattern.
- Documents can be traversed with XPath queries instead of adherent positions. Document parts are described in XPath 3.x, the Xml natural way to describe Xml document paths.
- Repositories. Document are retrieved and stored using the repository abstraction.

Why should we add a different approach?

The current Java binding tools usually follow an object/procedural approach on both the programming and the document side. The natural way should follow

an object/procedural approach on the object side and a document approach on the document side.

Why should we care about Xml/documents at all?

Documents are one of the most widespread ways of managing information. It has been like that since the early days of humankind. With the recent adoption of IT technologies, a new wave of programmers are more in favor of other other approaches, like records, objects, graphs, etc. Despite that, we create billions of documents every day, with office like and other tools, like operation receipts, reports, etc. Content is usually translated to Xml and Xml related formats like html, svg, etc. due to its unmatched capability to handle metadata, presentation instructions and mixed content. Even programming code is usually defined with some sort of document like structures (program code files).

We are not close to abandon documents soon and Xml is the most usual coding implementation due to the ability to naturally blend information, metadata, presentation indications, etc.

2. Introducing Fento.

Fento is a Java prototype/framework for object-Xml/document bindings. The first, obvious, characteristic is that mapping information is managed with Java Annotations.

Annotations are a commonly accepted configuration technique in the Java context. It is more coupled than external files, like Xml or yaml, but easier to manage as the Java code are directly marked.

The following listing shares some sample bind annotations in Fento.

```
@Bind(expression = "/concert-list/shows/Concert")
public class Location implements Serializable {

    @Bind(expression = "./event/Band")
    private Group group;

    @Bind(expression = "./name")
    private String name;
```

Java annotations are also used in the most shared Object/Xml binding tool in Java (JAX-B standard). The main difference with JAX-B is that Fento annotations are not based on "positional logic". The mapping expression is based on XPath 3.x.

Mind that XPath 3.x may requiere a schema alias in the XPath expression, opposed to the usual Java XPath 1.x. You may require to use "//*:some-node-name" instead of "//some-node-name" for example. XPath 3.x it is also more powerful.

3. Documents, flexibility and complexity.

Documents are a flexible data storage option. It's not uncommon to author Xml documents by hand. Both authors and tools may add data to the same documents.

Authors may add specific information for later use by hand, and develop the data integration later on (or not), depending on the project fund availability or business needs.

Xml documents frequently have information based on combinations of several schemas at the same time. This is a common practice when different teams are responsible for different parts of the data domain. On large data domains, different groups can evolve his specific schema without forcing every one else to update at once. Documents can achieve a considerable level of complexity even with a limited number of data instances.

We can also use documents to store object serializations. Although when it comes to object serializations, JSON is probably more appropriate.

JSON is also flexible and directly intended for object serialization. It is meant to be used from systems, and its data structure is aligned with the usual programming data types (Maps, Arrays, Strings, integers,..).

A key difference with documents is that JSON is expected to be used from a running system. The limited JSON metadata capabilities are remediated by the information that can be retrieved from the systems that generate them. There are also techniques that can be used, like JSON Schema, Open APIs descriptors, etc. but JSON excel where simple data serialization is used, generated or consumed by live systems.

The consequence of being aligned with a running systems is that flexibility and complexity, that JSON is capable of, is constrained by the effort and cost of evolving those systems. Keeping several data structure versions at once, aligned between serializations and systems is challenging.

Semantic based markup instead, with element names and schema aliases, has an advantage when dealing with several slightly different variations of document shapes and element positions.

In order to navigate this complexity, variability and even uncertainty, XPath is key. XPath can be used to define complex queries, taking advantage of the inherent relations (like positions, ancestors, siblings, etc.) and multiple schemas. It is the shared approach for addressing elements in tools like Xslt, Xquery and others, the natural approach on the Xml context.

4. The serialization problem.

Xml is frequently used on big documents, where we may only be interested on a small part.

When we deserialize a complex document to an object structure and serialize it back again to a document or an object serialization, if we don't consider all the pre-existing elements we will loose information. To bind the document structure to an object we will also need to know the format in advance and detail, which may usually not be the case.

Fento takes a more document like orientation, we don't describe binding based on structural positions or require a similar object structure to be prepared. Fento binding can focus only on the meaningful nodes, traverse the Xml document tree with XPath expressions and retrieve only the data that we expect to use.

The following listing shows a POJO marked (part) with binding annotations:

```
@Bind(expression = "/*:document-content")
public class Spreadsheet1 implements Serializable {

 @Bind(expression="..//*:table-row[4]/*:table-cell[3]/*:p/text()")
 private String name;
 @Bind(expression="..//*:table-row[5]/*:table-cell[3]/*:p/text()")
 private String surname;
 @Bind(expression="..//*:table-row[6]/*:table-cell[3]/*:p/text()")
 private String age;
```

The following operation would be used to retrieve document information and place it in a the pojo files:

```
FilePathRepository repository =
new FilePathRepository(CommonTest.getBasePath()+"templates/
spreadsheet1.xml");
Spreadsheet1 spreadsheet = repository.load(new Spreadsheet1());
```

Once the final data is obtained we can update only those nodes on the original document. We are not going to retrieve and rebuild the full document structure, where any mistake could end on lost data.

Once the POJO data is updated we can serialize it with the following command:

```
repository.persist(spreadsheet);
```

5. The repository concept.

Java persistence API relies on the repository abstraction concept. Fento tries to ease the document retrieval and storage process by also adopting the repository abstraction.

Fento provides several repository implementations from scratch:

- Disk File/Folder based repository ("FilePathRepository"), the simplest and the most obvious one.

- HttpRepository. A repository for sending and retrieving documents from an HTTP server.
- ClassPathRepository. Useful for accessing files on the application classpath, like configuration resources.
- ZipArchiveRepository. Since there are a lot of Xml resources in zip based files, like open document text or spreadsheets.

File pattern adjustment.

Most of the repository implementations support some sort of document path adjustment, based on the specific object properties. We could define a file path repository like this:

```
FilePathRepository repository = new FilePathRepository("C:/collections/
shows/concerts-{id}.xml");
```

When we try to retrieve or persist a document, the repository would adjust the path based on the object "id" property. Any of the object properties can be used for this purpose, separating object composition using dots.

6. A Repository in action.

Let's take a look at the class path repository. In Java applications we use to locate configuration files by classpath. In many cases we can't preview the application location (or locations), but configuration files provided as part of the application can be located by the class path relative placement.

The most usual case is the log4j.xml file. The following example show the marked DTOs/Pojos for a log4j file. The main class will be marked like this:

```
@Bind(expression = "/*:configuration")
public class LogLevel implements Serializable {

  @Bind(expression = "../*", classNames =
"net.vionta.xmlrepository.test.beansamples.log.Appender")
  private ArrayList<Appender> appender = new ArrayList<Appender>();

  public LogLevel() {
     super();
  }

  public ArrayList<Appender> getAppenders() {
   return appender;
  }
```

The class has a list of appenders that math the log4j appenders configuration. The appender class significant code is shared below:

```
@Bind(expression="*:appender")
 public class Appender implements Serializable {

 @Bind(expression = "@*:name",key=true)
 private String name;

 @Bind(expression = "*:param[@name='Threshold']/@value")
 private String level;

 public Appender() {    }

 public String getName() {
 return name;
 }
```

The code needed to retrieve the log4j information is shown below:

```
LogLevel logLevel = new LogLevel();
 ClassPathRepository repo = new ClassPathRepository();
 repo.setPath("log4j.xml");
 logLevel = (LogLevel) repo.load(logLevel());
```

From there, we can explore the log4j appenders levels using the DTOs adjust the level based on the appender name, and store the information again with the following command:

```
repo.persist(logLevel);
```

Check on the Fento repository examples for more detail about the operation.

7. Easing document management.

One of the most usual cases where we face complex document interactions is with office documents. Open office document format relies on zip wrapped files. We are going to explore a spreadsheet case and a text document case.

Reading and updating spreadsheet information.

We are going to query and update the following spreadsheet:

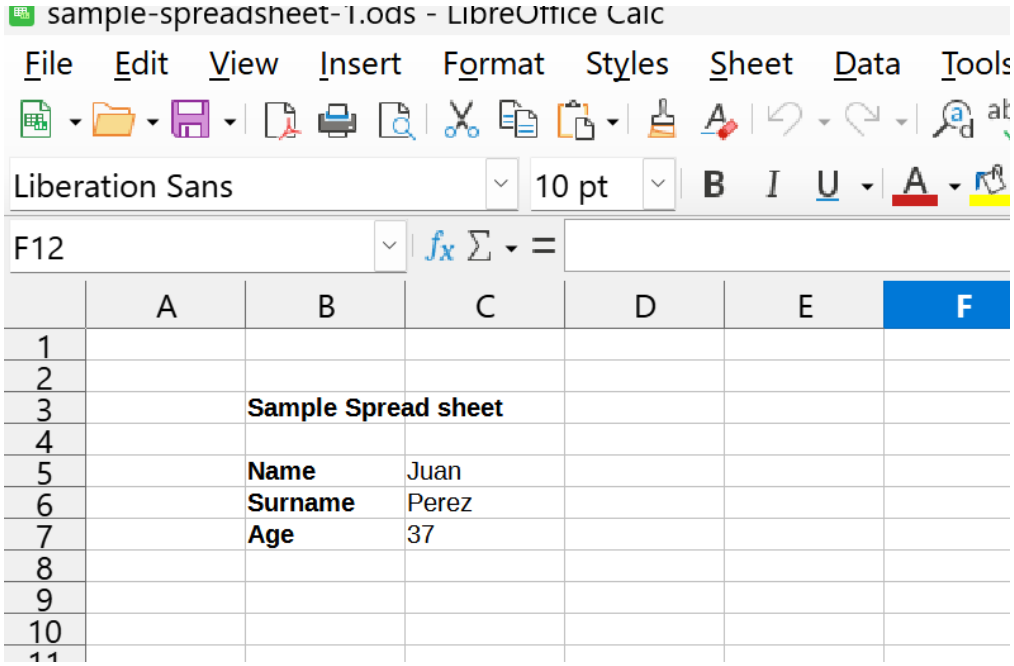


Figure 1. Open Spreadsheet Previous

The DTO/mapping code can be checked below:

```
@Bind(expression = "/*:document-content")
public class Spreadsheet1 implements Serializable {

    @Bind(expression="..//*:table-row[4]/*:table-cell[3]/*:p/text()")
    private String name;
    @Bind(expression="..//*:table-row[5]/*:table-cell[3]/*:p/text()")
    private String surname;
    @Bind(expression="..//*:table-row[6]/*:table-cell[3]/*:p/text()")
    private String age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
    this.name = name;
    }
    public String getSurname() {
      return surname;
    }
    public void setSurname(String surname) {
    this.surname = surname;
    }
    public String getAge() {
     return age;
    }
    public void setAge(String age) {
```

```
        this.age = age;
      }

      public String toString() {
        return "Spreadsheet1 [name=" + name + ", surname=" + surname + ",
age=" + age + "]";
      }

  }
```

The repository commands needed are shown below:

```
// Loading the spreadsheet data
ZipArchiveFileRepository repo = new ZipArchiveFileRepository();
repo.setZipFilePattern(CommonTest.getOutputPath()+"openDocument/sample-
spreadsheet-1.ods");
repo.setFileNamePattern( "content.xml");
Spreadsheet1 spreadsheet1  =  repo.load(new Spreadsheet1());

// Updating the spreadsheet information
spreadsheet1.setAge("55");
spreadsheet1.setName("Iturralde");
spreadsheet1.setSurname("Ruiperez");

// Persisting the information
repo.persist(spreadsheet1);
```

Note: Check that all your spreadsheet fields are text based to avoid the need to update extra attributes.

After that operation the updated spreadsheet shows the following data:

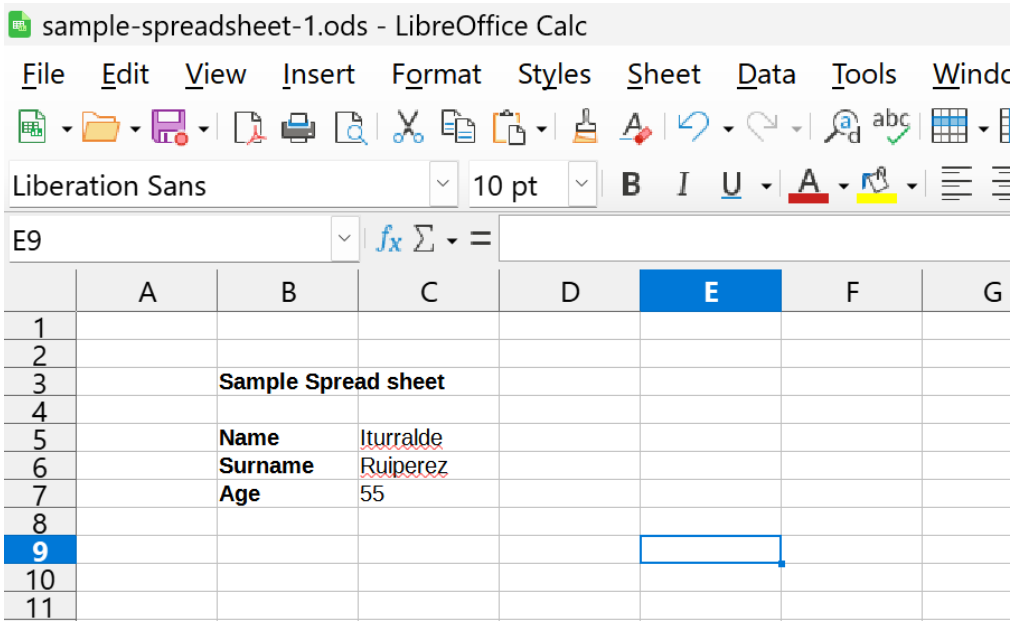


Figure 2. Open Spreadsheet Post

8. Updating an open document text file.

We can prepare a odt document for templating like this.

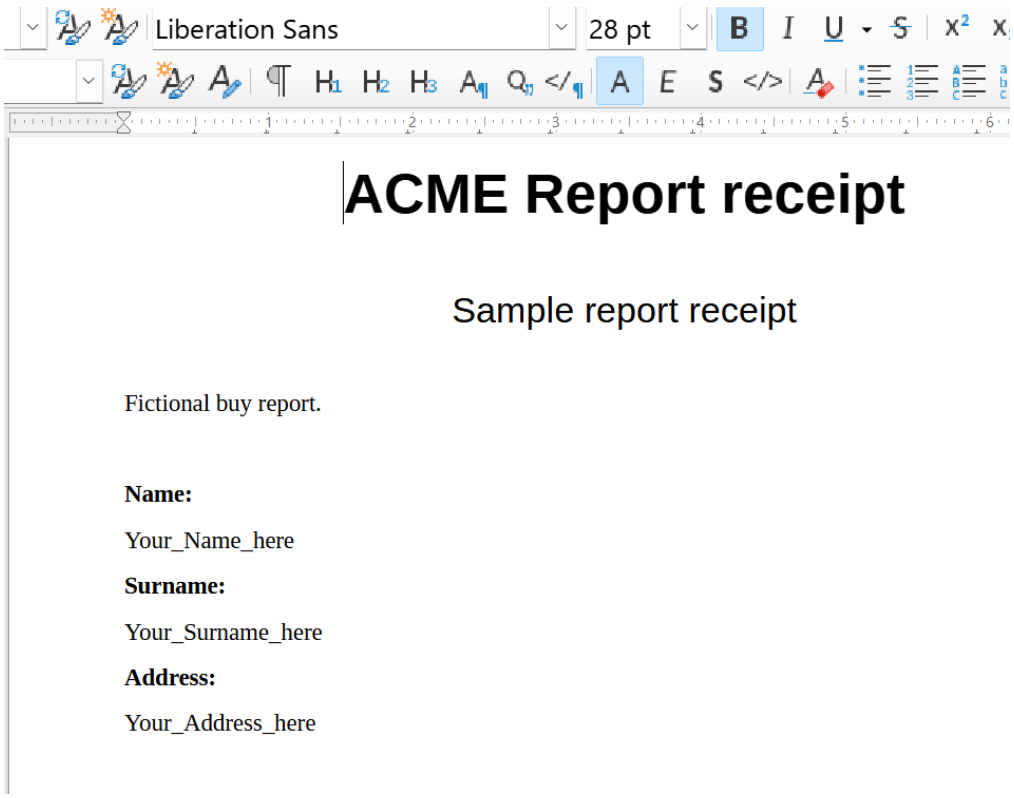


Figure 3. Open document text Pre

The code for the mapping class is:

```
@Bind(expression = "/*:document-content")
public class Document1 implements Serializable {

@Bind(expression="..//*:p[text()='Your_Name_here']")
private String name;

@Bind(expression="..//*:p[text()='Your_Surname_here']")
private String surname;

@Bind(expression="..//*:p[text()='Your_Address_here']")
private String address;

public String getName() {
  return name;
}
```

The code for updating the information is:

```
// Configuring the repository
ZipArchiveFileRepository repo = new ZipArchiveFileRepository();
repo.setZipFilePattern(CommonTest.getOutputPath()+"openDocument/text/
fictional-report.odt");
repo.setFileNamePattern("content.xml");

//Retriving the information
Document1 doc1 = repo.load(new Document1());

// Updating the information
doc1.setName("Jacinto");
doc1.setSurname("Gutierrez");
doc1.setAddress("Paseo de extremadura,15.4° Izq");

// Persisting the information
repo.persist(doc1);
```

With that we get the end result:

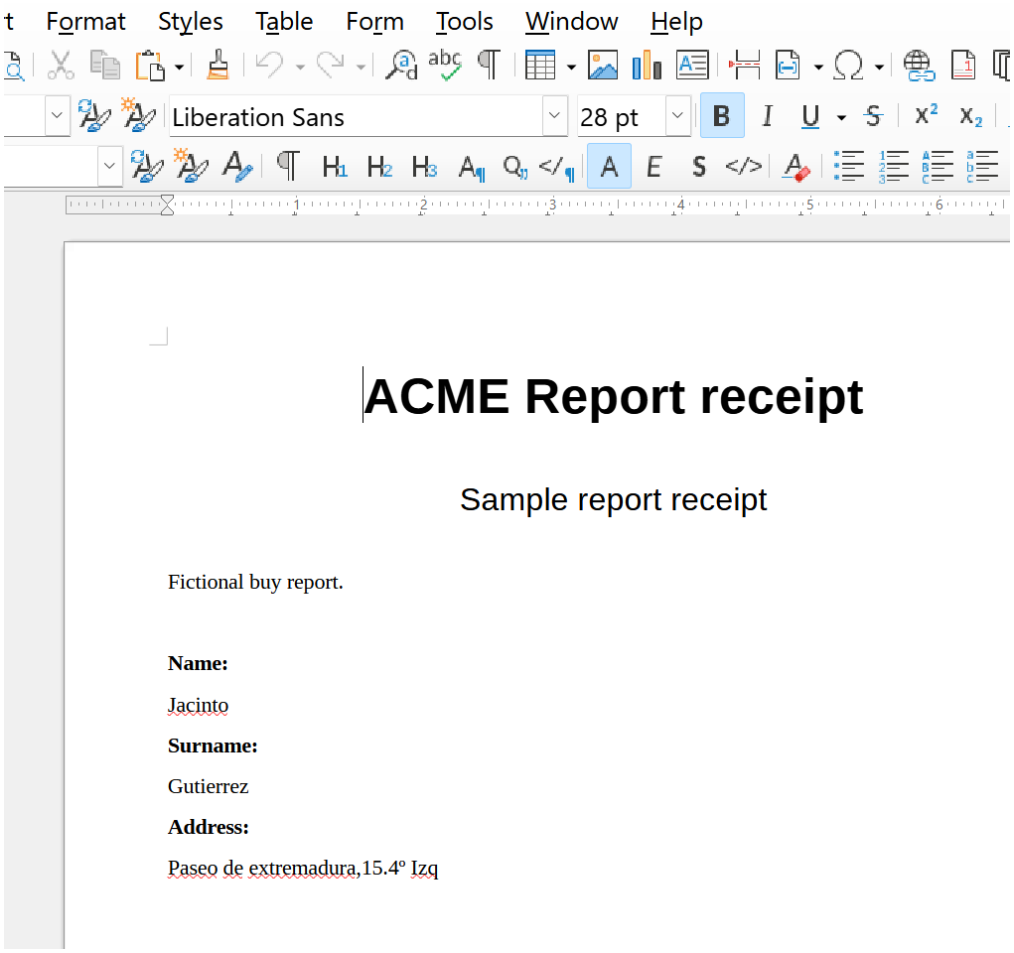


Figure 4. Document after modification

9. A safe approach for receiving and sharing information.

Xml/Document based information in occasions is too complex for direct interactions with external entities. If the interfaces are secured correctly, the approach may be more complex than the JSON equivalent and have no added benefits.

Java / DTO based applications are frequently used to implement safe interfaces. Technologies like JAX-RS (now Jakarta RESTful) or similar approaches that may use strong validation and a simple JSON based data model that exposes only the needed information.

Fento approach allows to combine document data with that kind of approaches. Internal information can be managed with documents, in a flexible manner, and expose some controlled amount of its information while the Java web framework may be used to limit the attack surface.

10. Criticism.

At this point, the proposal has some negative aspects that should be kept in mind.

Not viable for more complex use cases.

Cases where explicit navigation details, precise steps or specific behavior are needed won't fit Fento simple approach. We may, for example, map elements using axis and comparisons that are easy to read and retrieve but it could not be enough information to determine at which point we add new elements.

Still a lot of small cases to be defined and implemented.

Mapping instructions and options should still be studied and defined. There's a wide range of situations that may need adjustment especially when it comes to serializing and comparing collections of elements.

Provided code is a prototype, that needs revision and testing.

The provided sample is in the early stages of development, as an I+D project, and may need a strong revisions yet.

The document context difficults the adoption.

The current lack of appeal of Xml on the programmer groups may difficult the possibility of adoption.

Even there, we consider that the aproach should be useful for a wide range of cases. Easier to use than many competing options.

11. Conclusion

Most of the current object/Xml binding Java tools use an object approach for both objects and document sides. Fento balances the object approach on the Java side with a more flexible document approach on the Xml part.

Document management is a great way to handle complex/flexible sources of information inside a secured perimeter. It is cost effective, specially in constrained environments, with a limited number of documents.

For Java developers, using POJOs and annotations for managing configuration is a natural approach with no adoption barrier. Java is one of the main choices to build secure REST style data interfaces on main/corporate systems.

Removing the need to develop and map similar Java structures to complex documents and focusing on the meaningful parts of the information eases the document management process. Enabling Xpath 3.x, the natural Xml tool to query those document parts, opens the possibility of traversing the document in more precise and useful paths.

In general the proposed approach eases the document/Xml management for Java developers while preserving document querying flexibility on the document side. It can provide an easier tool for several common use cases.

Bibliography

[1] *XPath Standard* `https://www.w3.org/TR/xpath-31`

[2] *Xml Standarad* https://www.w3.org/TR/xml

[3] *Saxonica* https://www.saxonica.com/html/welcome/welcome.html

[4] *Java Xml Binding* https://jakarta.ee/xml/ns/jaxb/ https://javaee.github.io/jaxb-v2/ https://docs.oracle.com/javase/tutorial/jaxb/intro/index.html

[5] *Eclipse Link Moxy, JAX-B implementation* https://eclipse.dev/eclipselink/documentation/4.0/moxy/moxy.html

[6] *Jacson Xml* http://fasterxml.com/projects.html

[7] https://github.com/vionta/fento

[8] *Java Document Object Model* https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Document.html

[9] *Java Simple API for Xml* https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html

[10] *Java Streaming API for XML* https://docs.oracle.com/javase/tutorial/jaxp/stax/using.html

[11] *XStream Java to XML Serialization* https://x-stream.github.io

[12] *Apache CXF Aegis* https://cxf.apache.org/docs/aegis-21.html

[13] *JiBX: Binding XML to Java Code* https://jibx.sourceforge.io

XML Differencing Engine

Hauke Brandes
parsQube GmbH
<hauke.brandes@parsqube.de>

Nico Kutscherauer
data2type GmbH
<kutscherauer@data2type.de>

Liam Quin
Delightful Computing
<liam@delightfulcomputign.com>

Abstract

Comparing two XML documents and reporting differences between them is a difficult problem. The best solution depends on the context or environment, and sometimes, within that, the situation.

The authors of this paper evaluated existing research, and, not finding an existing solution whose architecture met the identified needs, decided to write yet another XML diff program, heavily influenced by the paper "Change Detection in Hierarchically Structured Information."

The present approach is unusual in two main ways.

First, it is written in XQuery.

Second, where the aforementioned paper proposes a "matcher" or comparison function that identifies corresponding pairs of nodes in the two trees being compared, the present work extends this concept: instead of a single matcher function, a pipeline of different matchers is used.

Each matcher uses a different algorithm to identify corresponding nodes. A matcher can focus on specific nodes, or can even correct or extend the result of previous matchers. The matchers used and the order in which they are applied can be chosen at runtime, and additional matchers can easily be added. Matchers are written in XQuery, although they could also be XQuery wrappers around XSLT stylesheets or even Java.

This architecture allows, for example, use of external information in determining how documents have been changed, and of different matcher pipelines for different types of document.

The result of running the program is an annotated version of one of the input documents which can then be converted into the other document; a side-by-side text diff is also supplied, and other visualizations may follow.

Keywords: XML, diff, changes

1. Introduction

This section gives some context and background to the project.

1.1. A Brief History of the Project

The Federal Office for Information Security (BSI) in Germany publishes highly complex Technical Guidelines (TR) for national and international cyber, identity security and standardization of information technologies. To meet the precision requirements of publication processes, the BSI, in a strategic partnership with the XML service provider data2type, developed a DocBook Toolchain (DBT) based on single-source publishing. This command-line tool validates and transforms DocBook documents (with a restricted vocabulary) via XSLT into output formats such as accessible PDFs and HTML, as well as Word/DocX for review processes.

When updating guidelines, there is a strict obligation to highlight the changes made to the documents. Historically, the BSI relied on the integrated Word comparison tool by comparing the generated Word target documents. However, this led to many errors and a "false positive inferno"—because the DocX documents were generated independently of one another, many structures were marked as changed due to invisible structural deviations (e.g., generated ID attribute values), even though nothing had actually changed. In addition, helpful meta-information (e.g., section IDs) could not be utilized during the comparison, as they only existed in the source files but no longer in the Word output. Consequently, extensive manual post-processing of the comparison results became necessary, and the overall quality suffered.

1.1.1. Evaluation and Market Analysis

To optimize this process and fully exploit the potential of media-neutral document maintenance, data2type, together with its partner company ParseQube, conducted a market analysis. The result was clear: existing tools were unsuitable for the given requirements.

- **Commercial solutions** (such as DeltaXignia or Oxygen XML) offered good general algorithms, but were ruled out for missing some important features. Specifically, they lacked the necessary adaptability to handle the semantic peculiarities of BSI-specific DocBook extensions and did not detected moved content without the existence of explicit ID anchors.
- **Open-source tools** (such as `xmldiff`) simply failed due to a lack of stability and reliability.

1.1.2. Solution Architecture: The Differ Project

Consequently, the authors were asked to develop a custom solution. By following the principle of Separation of Concerns they divided the system into three isolated components:

1. **The Core Differ Engine (Open Source):** A generic XML differ, controllable via configurations, that is strictly focused on comparing isolated XML graphs. It receives two XML documents, compares them, and returns an XML document as a result, which essentially corresponds to one of the two documents but carries the differences to the other via PI (Processing Instruction) annotations.
2. **The Diff Representation (Extension of the DBT):** The existing PDF output of the DBT was extended so that change annotations in the format provided by the Differ engine can be rendered in the page-based layout.
3. **The Logistics Layer (DBT-Diff):** An internal command-line tool that organizes the BSI workflow (Git checkout, XInclude resolution) and prepares the data for the diffing process to improve the performance. The biggest performance improvements here is the **ID-based splitting**: A large XML document is split into small fragments based on chapter IDs. Via ID comparisons, corresponding chapters are identified, and only these are compared with each other by the Diff engine. This drastically reduces the workload of the Diff engine. The results of the compared fragments are then merged back into a complete result and handed over to the DBT for rendering.

While the DBT and DBT-DIFF components are closely customized to the requirements of the BSI, the Core Differ Engine was designed strictly as a generic application. This differ can be reused in other projects, and it is planned to release it as an open-source project once the main development work has been completed. This paper will focus on the unique approaches and innovative ideas that were applied in the development of this engine.

2. Requirements: What It Does

The Differ program takes two input files representing an XML document in two different edit states.

Its result is an annotated version of one of the input documents; the annotations could be used by an external application to generate one document from the other, or to visualize the changes, for example in HTML of PDF.

Note that producing PDF or other format visualizations of changes is *not* part of the differ: rendering an XML document into PDF requires knowledge of the required formatting. However, enough information is available in the output that this can be done, for example with XSLT and then CSS or XSL-FO.

3. Prior Art

There have been quite a few attempts to make a useful, fast, generally applicable XML diff program, both open source and proprietary. A paper given at an earlier XML Prague conference introduced "Natural Diff" ([2]) which was written in Rust and intended to be open source, but its funders withdrew it and kept it proprietary.

There have been many others, including work by DeltaXignia (previously known as DeltaXML).

A paper that inspired this work ([1]) used structure, but did not use the semantics of element names. In addition, the paper relied on a "comparison function" to decide whether two nodes should be considered to correspond to one another in the documents being compared, but did not give such a function.

As far as the authors of the present paper are aware, the XML diff program here is the first to have an open architecture, in which comparison functions can be added at runtime, as described below.

4. How It Works

The differ has several separate phases, mostly written in XQuery.

First, the document is cleaned of irrelevant nodes (whitespace text nodes, comments) and normalised (e.g. consecutive spaces are reduced to a single space).

Then, a *matcher* is run on corresponding sections, or on the whole document, to identify elements that correspond to each other in the two inputs.

Once every node has been marked as a change, insertion, deletion, or as unchanged, a copy of the "newer" input is produced including annotations (in processing instructions) to mark the changes.

Finally, output is an XML document annotated with processing instructions, using the same conventions used in the Oxygen XML Editor, extended to support move as well as insert and delete.

5. Phases of the Diff Process

The entire diff process is divided into three consecutive main phases:

1. **The Matcher:** First, the nodes of both input documents are compared to find corresponding node pairs. If no partner can be found for a node, this results in an incomplete pair, which implicitly marks a deletion (node only exists in the first document) or an insertion (node only exists in the second document).
2. **The Edit Script Maker:** Based on the match result, a script is generated that describes the transformation of the first document into the second by means of a series of consecutive actions.

3. **The Edit Script Patcher:** Finally, the edit script is applied to the first input document. However, the actions are not executed destructively; instead, they are logged via a change-tracking mechanism using annotations in the document. The final result is an annotated variant of the second input document with change markups, which highlights the differences compared to the first document.

6. Running the XML Differ

Currently in production the program is integrated into an Ant pipeline, by a custom Ant task which is provided by the project. In future this can be easily extended by providing a standalone application for instance.

The Matcher component can also be run from the command-line or from within BaseX.

7. The Matcher in more detail

The purpose of the matcher is to mark nodes in each document with a reference to the corresponding node in the other, if there is one. The default output is actually a sequence of arrays. Each array contains two items, each item being either a map representing a match or an empty sequence. The following listing shows one such array:

```
[ (), {
  "node": <p>, an eminent English lawyer ... </p>,
  "matched": (),
  "type": "inserted",
  "why": "text value only in document 2",
  "confidence": 0.2
} ]
```

The first item in the example is an empty sequence, indicating that no corresponding node was found in the first document, so, this represents an insertion (the actual text has been abbreviated for this example). Since there is no corresponding node, the *matched* value is empty. The *why* field helps with debugging matchers.

The *confidence* field in the second item in the example merits further explanation. Each matcher in the pipeline, when it finds a correspondance between two nodes, assigns that correspondence a *confidence factor*; if the nodes have already been matched (either to one another or to other nodes) with higher confidence, the new match is rejected.

When the first item in an array in the result sequence is a populated map, and the second is empty, a deletion is represented.

When both items are populated maps, a match has been found between the two nodes.

An XQuery (or XSLT) application using this result receives actual document nodes here, and hence can process the documents; the edit script generator that is part of the XML Differ uses this output to insert annotations as processing instructions, as explained below. It is also possible to apply the matcher pipeline as a standalone query, using *use-matcher.xq* with external variables for original and changed documents, and, optionally, a comma-separated list of matchers to use, and, optionally, an action to perform other than the default (for example, listing all unmatched nodes, which ideally will be the empty set).

8. The Matcher Pipeline

The "matcher pipeline" probably the most unusual part of the XML Differ. It is a user-specified sequence of functions each of which uses a different algorithm or heuristic to identify corresponding nodes between the two documents being compared. These functions can be very fast (as XQuery processes go), generally walking one or both documents.

The matchers supplied understand about marked-up documents. For example, if a *section-like* container element has an *xml:id* or *id* attribute, and there is another section-like container in the other document with the same *xml:id* or *id* value, the section elements themselves are marked as corresponding, and their children are investigated to see if they, too, correspond, perhaps by text value. Further, if the two sections are the same (according to the *deep-equal* function) they are marked as identical, and subsequent matchers do not need to explore them further. This is done by the "id matcher" currently using a built-in list of elements; the function can easily be replaced by registering a different matcher function, and in the future the authors plan to make the list of elements be configurable.

8.1. Registering a Matcher Function

Currently, matcher functions are registered by editing *matcher-framework.xqm* although in the future a configuration file is planned. The matchers are simply listed in a map:

```
declare variable $mf:matcher-definitions as map(*) := map {
  "find-pairs" : map {
    "matcher" : m:find-pairs#3,
    "about" : "finds paragraphs with identical content"
  },

  "id" : map {
    "matcher" : m:match-by-id#3,
    "about" : "elements with the same id are matched"
  },
```

```
  "sections" : map {
    "matcher" : m:matcher-sections#3,
    "about" : "matches containers based on what their children match"
  },
  [. . .]
```

Here, each *matcher* is a function item (or it could be an inline function) and the *about* entries are produced if *use-matcher.xq* is called with the external variable *$matchers* set to the value *list*.

Matcher functions can be associated with any namespace, of course; if they are defined in an external module, this must be imported by *matcher-framework.xqm*; a future version of the Matcher will use a configuration file and may be able to load modules dynamically.

8.2. Some example matchers

This section describes three of the match functions included with the XML Differ. We will begin with the source code for the *id* matcher, then describe two others in less detail.

8.2.1. The ID Matcher

The purpose of the id matcher is to mark elements as matching if they have the same XML ID value, and, further, to mark the elements and their descendants as identical if they are deep-equal.

Of course, it is arbitrary as to whether ID values indicate correspondance; in some documents they are automatically generated, for example, and should be ignored. For such documents it would be best for the user not to use the ID matcher.

The following listings show the ID matcher in three parts. First the declaration:.

```
(:~
 : m:match-by-id
 :
 : A matcher that uses xml:id or id attributes to match nodes.
 : xml:id is preferred over id if both are given, assuming that
 : id is probably not actually an ID value in that case.
 :)
declare function m:match-by-id(
  $matchbox as mapi:matchbox,
  $tree1 as node()*,
  $tree2 as node()*
) as mapi:matchbox
{
```

```
  . . .
};
```

A *record type* is used, the *matchbox*, to represent the map containing the accumulated matches; this is taken from the XQuery 4 draft, is implemented by BaseX, and provides better error checking than simply using a map type. The *mapi* prefix is short for *matcher API*, a module of convenience functions to insulate matcher authors from the actual data representations of various types of object. The *match-by-id* matcher here will use the matcher API to add matches it finds to the matchbox, or, more precisely, will make a new matchbox object containing the new matches, and return it.

Now we should show the insides of the matcher, in two parts. First, it makes a map that stores ID values and corresponding nodes in the second of the two documents. To do this, it uses the Visitor Pattern [3], which will visit every element in the tree and call the given function, accumulating the result in the map:

```
(: first make a map of id to node in tree2: :)
let $idmap := map:remove(
  v:visitor(
    $tree2,

    map {
      $v:element : function(
        $node as element(*),
        $map as map(*)
      ) as map(*)
      {
        let $id := ($node/@xml:id, $node/@id)[1]
        return
          if (empty($id)) then $map
          else map:put(
            $map,
            xs:string($id),
            $node
          )
      }
    }
  ),
  $v:element
)
```

The matcher then walks the *first* tree, this time looking for elements with @xml;id or @id attributes whose values are in the constructed *$idmap*:

```
return v:visitor(
  $tree1,
```

```
  map {
    "m" : $matchbox,

    $v:element : function(
      $node as element(*),
      $map as map(*)
    ) as map(*)
    {
      let $id as xs:string? := ($node/@xml:id, $node/@id)[1]
      return if (empty($id)) then $map
      else if (empty($idmap($id))) then $map (: id not in doc 2 :)
      else let $othernode := $idmap($id)
      return if (deep-equal($node, $othernode))
      then map:put(
        $map,
        "m",
        mapi:make-match-object(
          $map?m,
          $node,
          $othernode,
          "identical", (: match type :)
           "same id value and deep-equal",
          1.0 (: confidence :)
        )
      ) else if ($othernode)
      then map:put(
        $map,
        "m",
        mapi:make-match-object(
          $map?m,
          $node,
          $othernode,
          "match",
          "same id value",
          1.0
        )
      ) else $map (: no new match in this case :)
    } (: function :)
  } (: map :)
)?m (: visitor :)
```

The *mapi:make-match-object()* function returns a copy of its input "matchbox" but with the new match added, if the confidence is higher than any existing match for the given two nodes. The code has been somewhat simplified, as the actual function processes descendents or children before marking the elements as matched.

8.2.2. The Sibling Matcher

The sibling matcher looks for gaps:

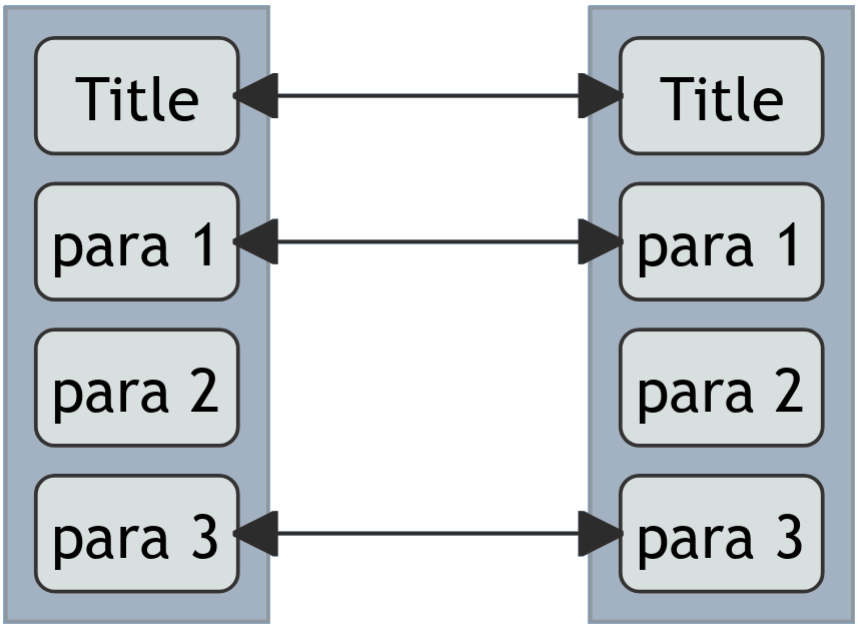


Figure 1. Sibling Matcher Scenario

In the figure, arrows indicate matches already determined; when the sibling matcher considers para 2, it will see that para 1 immediately before it, and para 3 immediately after it, are matched to elements surrounding a single para2, and will therefore join the two para 2 elements.

The confidence of the match is highest if the two "para 2" elements are deep-equal, lower if they have similar text and lowest of all otherwise. Text similarity is determined using a language-specific XQuery implementation of an algorithm for German stemming described by Jörg Caumanns [1], as modified for use in Apache Lucene and elsewhere, combined with unweighted cosine vector similarity [4].[2]

8.2.3. The Section Matcher

This matcher could more accurately be called the Container Matcher. Its goal is to make sure that elements that contain paragraph-like elements or other containers are matched where possible.

This matcher chooses candidate sections by considering the match-ends of each matched paragraph within the section, and choosing the section with the

[2]The phrase "unweighted cosine vector similarity" here is a fancy way of saying number of stemmed words in common. The use of stemming reduces the effect of changes that affect word endings, for example.

most match-ends. The Figure 2 illustrates this, with the leftmost and uppermost section elements being conssidered closest and the arrows indicating already-determined matches.

8.2.4. The Paragraph Similarity Matcher

The paragraph similarity matcher is based on the temporary matching result of the previous matchers. Its goal is to find partners for the remaining, unmatched paragraph-like elements (hereinafter simply referred to as paragraphs) that are strongly similar in terms of text content.

A naive approach, where every unmatched paragraph of the base document is compared with every unmatched paragraph of the target document, would lead to exponential effort and drastically degrade performance. Additionally, this could lead to match results where paragraphs have been unexpectedly moved around the document. To minimize the number of comparisons and respect the context of the paragraphs, an attempt is made upfront to identify paragraphs that occupy similar structural positions in both documents. For this purpose, the paragraphs are sorted into manageable groups of comparison candidates (sequence pairs) in a two-stage process.

8.2.4.1. Stage 1: Container Projection

In the first stage, the existing container matches are used to make rough spatial assignments. First, all match pairs are determined that are not paragraphs themselves and are also not descendants of paragraphs (this excludes inline matches). What remains are the structural containers.

Subsequently, each paragraph is assigned to the match pair that is found first on the ancestor axis. If a paragraph has no matched ancestor, it is assigned to a so-called *null group*. Since this projection is performed for both documents, the container pairs give rise to so-called *sequence pairs*, which contain paragraphs from both documents. This ensures that in the subsequent processes, only paragraphs that are located within the same logical container are compared. Note: At this point, the sequence pairs also contain paragraphs that have already been matched. These will be used in Stage 2 for further fragmentation.

8.2.4.2. Stage 2: Formation of Sub-Sequence Pairs via LCS

In the second stage, each sequence pair is considered in isolation. The goal is to further subdivide the group into *sub-sequence pairs*.

1. **Filtering out external matches:** First, all paragraphs are ignored that already have a match partner, but which is *not* located in the same sequence pair. These paragraphs were obviously moved to a different container and play no role in the local group formation.

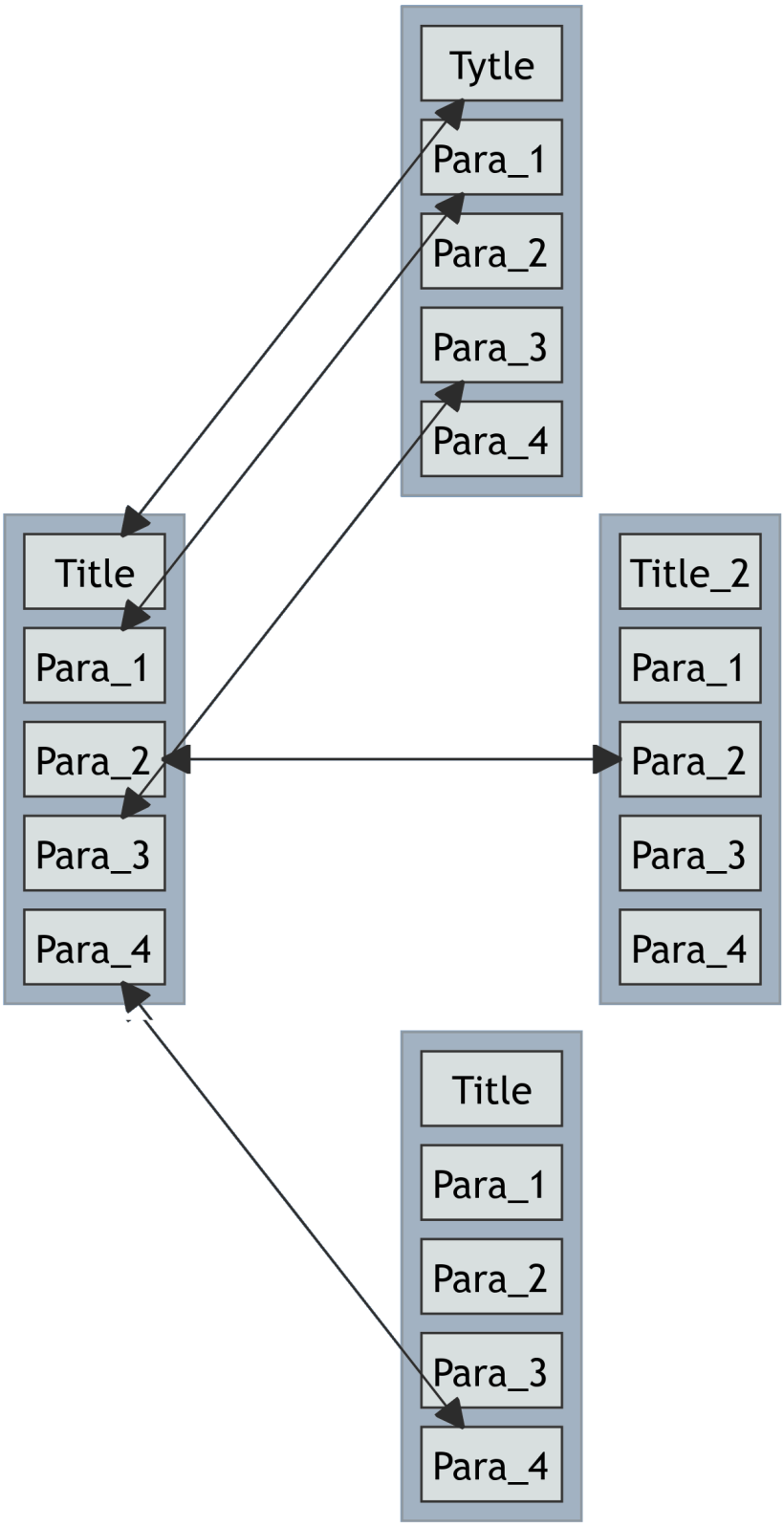


Figure 2. The Section Matcher

2. **LCS Determination:** The remaining paragraphs with a match partner in the same sequence pair are now considered in their respective document order. Since the order in the base and target documents can differ (crossovers), the *Longest Common Subsequence (LCS)* of these internal matches is determined using an XQuery implementation of the Myers algorithm ([6]). Pairs that are not part of the LCS are considered local moves and are likewise filtered out for the group formation.

3. **Assignment of unmatched paragraphs:** The remaining LCS match pairs now form the anchor points of the applied fragmentation. Each paragraph *without* a match partner is assigned to the LCS match pair that follows next in the document order. If an unmatched paragraph is at the very end (after the last LCS match pair), it is assigned to another *null group*. This is applied to both sequences of the pair, so that each pair and the null group to which at least one paragraph has been assigned forms a sub-sequence pair.

This process results in very fine-grained sub-sequence pairs. Only the unmatched paragraphs within the same sub-sequence pair are checked for text similarity in the subsequent step.

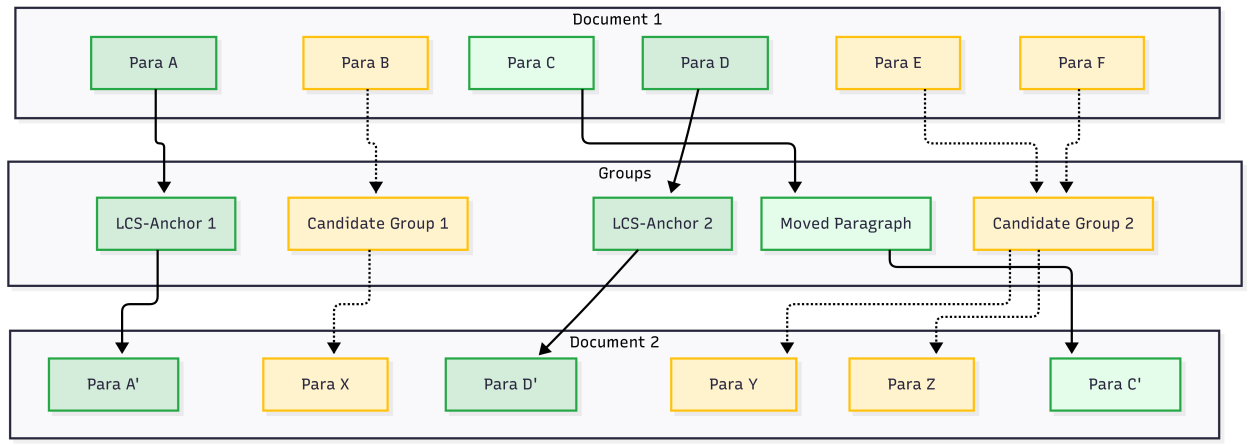


Figure 3. Group Formation based on LCS Anchor Pairs

In the figure the matching pairs `[Para A|Para A']` and `[Para D|Para D']` belong to the *Longest Common Subsequence*. Consequently, the pair `[Para C|Para C']` is considered to be moved. In Document 1 the unmatched paragraphs are `Para B`, `Para E` and `Para F` while in Document 2 they are `Para X`, `Para Y` and `Para Z`. Since the LCS anchor pair `[Para A|Para A']` has no preceding unmatched paragraphs, they do not form a sub-sequence pair. `Para B` is assigned to `Para D`, as is `Para X` to `Para D'`. So the first sub-sequence pair is `Para B`, `Para X`. The *null group* contains the remaining unmatched paragraphs: `Para E`, `Para F` (from Document 1) `Para Y` and `Para Z` (from Document 2).

8.2.4.3. The Text Similarity Comparison

The actual similarity check takes place within the isolated sub-sequence pairs using the XQuery function `psim:find-similars`. It is declared as follows:

```
declare function psim:find-similars(
  $global-map as mapi:global,
  $paras_1 as node()*,
  $paras_2 as node()*,
  $threshold as xs:double := 0.6,
  $length-tolerance as xs:double := 0.2
) as mapi:global
```

The parameters `$threshold` and `$length-tolerance` are used for configuration. They determine how strongly two paragraphs must resemble each other to even be admitted as comparison candidates (length tolerance) and to ultimately be accepted as a match (similarity threshold).

The function executes the following algorithm:

1. **Iteration:** Iterate sequentially through the sequence of paragraphs from the base document (`$paras_1`).
2. **Filtering candidates:** For the current paragraph, find all remaining available paragraphs from the target document (`$paras_2`) that have a similar text length. The maximum permissible deviation is determined by the factor `$length-tolerance` (applied to the current paragraph). This forms the pool of comparison candidates.
3. **Sorting candidates:** Sort these comparison candidates in ascending order by their absolute text length difference to the current paragraph (the most similar in terms of length comes first).
4. **Similarity check:** Find the first candidate in the sorted list that exhibits sufficient textual similarity.
   - For this, the same function for calculating text similarity is used that is also employed in the Sibling Matcher (see Section 8.2.2).
   - The comparison function returns a value between `0` (completely different) and `1` (identical).
   - If this value is greater than `$threshold`, the candidate is considered similar and a new match pair is found.
5. **Finishing the iteration:** If a match is found, the successful candidate is removed from the pool `$paras_2` so that it cannot be matched again. Subsequently, the process continues at step 1 with the next paragraph from `$paras_1`. If no candidate reaching the threshold was found, the current paragraph remains unmatched and the process also continues with the next one.

8.2.5. The Inline Matcher

The Inline Matcher is designed to be executed as late as possible in the matcher pipeline. It assumes that similar paragraphs (or paragraph-like elements) have already been matched to each other by preceding matchers (such as the paragraph similarity matcher). Its primary task is to determine the detailed, inline changes (changes at the word and character level) for these similar, but not identical paragraphs.

8.2.5.1. Extension of the Matcher API (Snippet Matches)

To represent inline changes, the previous node-based matching logic was no longer sufficient. The matcher API was specifically extended for this type of matching, so that not only entire nodes can be matched to each other, but also exact sub-regions within a text node value. For this purpose, the following function was added to the API:

```
declare function mapi:make-value-match-span (
  $map-so-far as mapi:global,
  $node1 as node()?,
  $node2 as node()?,
  $type as xs:string,
  $reason as xs:string,
  $confidence as xs:double,
  $start1 as xs:integer?,
  $start2 as xs:integer?,
  $length1 as xs:integer?,
  $length2 as xs:integer?
) as mapi:global
```

This function behaves very similarly to the regular function `mapi:make-match-object`, but has three crucial differences:

1. **Sub-string addressing:** In addition to the matched nodes, a starting character (`$start1`, `$start2`) and a length specification (`$length1`, `$length2`) are passed in each case.
2. **Multiple matches per node:** Unlike the node-based function, this function allows multiple matches for the same text node, as long as they do not overlap. If overlaps occur, the `$confidence` value determines which match is kept.
3. **Extended match object:** The match object, which is stored in the match box for the matching result, now also contains the position information of the snippet.

Example 1. Structure of a Snippet Match (Deletion of a Word)

In this example, the word "eminent" was deleted from the English sentence. The match object references the start index (6) and the length (7) of the deleted character string.

```
[
  map {
    "node" : text{, an eminent English lawyer ... },
    "matched" : (),
    "type" : 'deleted',
    "why" : "Closest ancestor are matched and the value fragments
matches in order",
    "confidence" : 0.768,
    "start" : 6,
    "matched-start" : (),
    "length" : 7,
    "matched-length" : ()
  },
  ()
]
```

8.2.5.2. Algorithm for Fragmentation and Sequencing

The Inline Matcher processes each already matched paragraph pair individually, performing the following steps:

1. **Fragmentation:** For both partners of the paragraph pair, all descendant text nodes are fragmented. This fragmentation should be controllable via configurations in future, but is currently only implemented for word boundaries. Here, each isolated word as well as every space between (whitespace, punctuation, etc.) becomes its own so-called *snippet*. Each snippet remembers (via a map) its text value, its origin node, and its starting position.
2. **Pseudo snippets:** For each empty descendant element (e.g., an `<xref/>` in DocBook), special pseudo snippets are created. Since a partner in the other document cannot be determined for them based on their textual content, they are treated as special content in the snippet sequence to be formed.
3. **Sequence formation:** This process results in two flat sequences of snippets per paragraph. The original order in which the snippets appear in the document is strictly preserved here.

8.2.5.3. Sequence Comparison via the Myers Algorithm

The two flat snippet sequences are now passed to the XQuery implementation of the Myers algorithm. Since these are not simple strings here, the Myers algorithm uses a dedicated `compare` function that compares two items from the input sequences and returns a boolean value:

- **Word snippets:** Only return `true` if the words are exactly identical (no fuzzy similarity check takes place at this level).

- **Whitespace snippets:** A normalization of the spaces is performed prior to the comparison.
- **Empty elements (pseudo snippets):** Are considered identical if the respective element name matches.
- **Mixed forms:** XML elements and text snippets are never considered identical.

Based on this comparison logic, the Myers algorithm returns a diff result in which snippet pairs are formed and the remaining items are explicitly marked as deletions or insertions. This diff result is used to generate the final snippet matches via the `mapi:make-value-match-span()` function.

8.2.5.4. Assignment of Inline Elements

After the text snippets have been matched, the Inline Matcher attempts in a final step to match the surrounding inline elements (which contain the corresponding text, e.g., emphases or links) to one another. Here, a match factor is calculated for all potential element pairs. The following criteria are factored into this calculation with varying weights:

- **Textual overlap:** How much textual content within the two elements has been matched to each other?
- **Name equality:** Do the XML element names match?
- **Relative position:** Does the position within the paragraph match (measured by the number of preceding inline elements in the same paragraph)?
- **Hierarchy level:** Does the hierarchy level match (measured by the number of ancestors that are still located within the same paragraph)?

The assignment process is iterative: the pair with the highest match factor is always selected and confirmed. This is repeated until the highest remaining match factor falls below a defined threshold. All unassigned inline elements remaining thereafter are marked as insertions or deletions.

9. The Edit Script Format

The Edit Script format is an XML-based format developed within the Differ project to formally describe the transformation of a document from an initial state to a target state. The project provides its own RelaxNG schema for this purpose.

Within the root element `<editscript>`, a list of consecutive actions is defined. A fundamental principle of the Edit Script is sequential processing: the actions are executed strictly one after the other. This means that the resulting output document of one action serves immediately as the input document for the next action.

9.1. Addressing via Node Index Paths (NIP)

To precisely address nodes (elements, text, or attributes) that are to be deleted or modified, the format dispenses with classic XPath expressions. Instead, so-called **Node Index Paths (NIP)** are used. These paths are specified in the actions via the `@select` or `@to` attribute. The basic principle is that each NIP must yield exactly one node. It is always an error if no node is selected - and by logic, it is impossible for more than one node to be addressed.

A NIP uses a highly simplified pattern (e.g., `/1/2/1/4/2`), where each number refers to the n-th child node in the hierarchy. This enables very fast and unambiguous navigation within the document tree. These are also found as "tumblers" in XPointer child sequences.

The only exception to this purely numerical pattern is the addressing of attributes. These are addressed at the end of the path using a URIQualifiedName (according to the [7] recommendation) prefixed with an `@`. Consequently, the pattern is `@Q{namespace-uri}local-name`. For example, `/1/3/@Q{http://www.w3.org/1999/xlink}href` refers to the `href` attribute in the XLink namespace of the third child node of the root element.

9.2. Action Types

The Edit Script format defines various basic actions to represent all conceivable changes to an XML tree.

- **Remove (`<remove>`)**: Removes the node specified via `@select` (by NIP) from the document.
- **Insert (`<insert>` and `<add-attribute>`)**: Inserts new nodes into the tree.
  - With `<insert>`, elements or text nodes are inserted. The path in the `@select` attribute designates the anchor node. The `@pos` attribute specifies where the new content is inserted relative to the anchor node (`before`, `after`, `first-child`, `last-child`). The new content is defined by the content of the action element.
  - The specialized element `<add-attribute>` is used for attributes and namespaces. Here, `@select` defines the target element. The new names and values are provided via the `@name` (and optionally `@ns`) attributes and the text content of the action. If the name starts with `xmlns`, a namespace node is added.
- **Move (`<move>`)**: A node is removed from its original position and reinserted elsewhere in the document. The node to be moved is addressed by the `@to` attribute, while `@select` and `@pos` (as with the Insert action) define the new target anchor node and the relative position.

Note: As an alternative to a direct `<move>`, movements can also be expressed by a separate pair of `<insert>` and `<remove>`. These are then logically linked to each other via an identical `@move-id` attribute (1:1 relationship). The order of these two linked actions in the script is irrelevant.

- **Rename (`<rename>`)**: Changes the name of an existing element. The affected node is addressed via `@select`, the new identifier is passed in `@name`, and the corresponding namespace in `@ns` if necessary.
- **Update Value (`<update-value>`)**: Changes the value of a text or attribute node. The addressed node is selected via `@select`. Using the optional `@start` and `@length` attributes, operations can be performed very granularly by replacing only a specific character string from a certain position, rather than necessarily the entire content.

9.3. Example

The following example demonstrates how an edit script sequentially transforms a simple XML document by applying different actions.

Example 2. Input Document

```
<doc>
    <title>Hello World</title>
    <p>Delete me</p>
</doc>
```

Example 3. Edit Script

This script renames the first child (the title), removes the second child (the paragraph), and adds an attribute to the root element.

```
<editscript xmlns="http://bsi.bund.de/ns/edit-script">
    <!-- Rename 'title' (/1/1) to 'header' -->
    <rename select="/1/1" name="header"/>

    <!-- Insert a 'p' after the title (/1/1) -->
    <insert select="/1/1" pos="after">
       <p>Add me</p>
    </insert>

    <!-- Remove the 'p' element (/1/3) -->
    <remove select="/1/3"/>

    <!-- Add a 'status' attribute to the root element (/1) -->
    <add-attribute select="/1" name="status">draft</add-attribute>
</editscript>
```

Example 4. Resulting Output Document

```
<doc status="draft">
    <header>Hello World</header>
    <p>Add me</p>
</doc>
```

10. Edit Script Generation

The edit script is generated based on the result of the preceding matching process and the two input documents (base document and target document). The result of this matching essentially consists of associated node pairs (one node from each document). Within these pairs, a partner can also be missing, which indicates a deletion (node is missing in the target document) or an insertion (node is missing in the base document).

10.1. The Core Algorithm

The generation of the concrete edit script actions is carried out by traversing these node pairs according to a strictly defined sequence:

1. **Initialization:** The starting point of the algorithm is necessarily the node pair consisting of the root nodes of both documents.
2. **Recursive Processing:** For a given node pair, the necessary actions to transform the base partner into the target partner are determined in the following order:
   a. *Identify child node pairs:* Find all respective child nodes for both partners and determine the match pairs among them.
   b. *Depth-First Search (Recursion):* Execute this entire step 2 recursively for each of these child pairs, specifically in the order they appear in the base document. The resulting actions are inserted into the edit script first (bottom-up approach).
   c. *Deletions:* Generate a `<remove>` action for all child nodes of the base partner whose match partner is *not* a child node of the target partner. Important: These actions are sorted such that the nodes are deleted strictly in the *reverse order* of their occurrence in the base document. This ensures that the index references (NIP) always refer to the unmodified state of the node and are not shifted by previously executed deletions.
   d. *Reorderings (Moves):* Compare the orders of the child pairs detected in step (a) in both documents. A *Longest Common Subsequence (LCS)* is determined. For each child node of the base partner that is not part of this LCS, a `<move>` action is generated to move it to the position of its partner in the target document.

e. *Insertions:* Generate an `<insert>` action for all child nodes of the target partner whose match partner is *not* a child node of the base partner. These are inserted in the exact order they appear in the target document, ensuring that the index references (NIP) always correspond to the indices in the target document and are not distorted by nodes that are still missing (inserted later).

f. *Element-specific adjustments:* If the currently considered node pair consists of two XML elements:

   1. Compare their attributes and, if necessary, generate actions to add, remove, or modify these attributes.
   2. Finally, compare whether both partners have the same element name. If this is not the case, a `<rename>` action is generated.

10.2. Detailed Analysis of the Phases

10.2.1. Deletions and the Preservation of Descendants (re Step c)

When a node is deleted, it does not necessarily mean that all of its descendants should completely disappear from the document. It can happen that descendants are preserved elsewhere through moves. Before the actual deletion action for the parent node is generated, all descendants intended to be moved are first deleted in isolation. This specific deletion is supplemented by a `@move-id` that is unique to this match. When the partner is later inserted at the new target position, the same `@move-id` is used to logically couple the actions.

Additionally, in this step, deletion actions are also generated for child nodes that do have a match partner, but this partner is not a child node of the target partner (i.e., nodes that have left the current parent node). These deletion actions also receive a `@move-id` unique to the match.

10.2.2. Determination of the Longest Common Subsequence (re Step d)

To determine the LCS, the sequences of child nodes are compared using the *Myers algorithm* ([6]). The Myers algorithm specializes in identifying the longest common subsequence (LCS) in differently sorted sequences of equivalent items by determining corresponding deletion or insertion operations for all remaining items (those not included in the LCS), which serve as the basis for the move actions here.

10.2.3. Insertions with Moved Descendants (re Step e)

Similar to deletions, there are also special considerations regarding descendants when inserting nodes. When a new node is inserted, it may be that some of its

descendants are not completely new, but are intended to be "moved" there from another location.

To represent this, the regular insert action of the parent node is executed first, but it initially only contains those descendants that are *not* affected by a move. Subsequently, separate insert actions are generated for the moved descendants, linked with the corresponding `@move-id`.

Analogous to step (c), insert actions are also generated in step (e) for child nodes that have a partner which is not a child node of the base partner (i.e., nodes that were moved into this parent node from elsewhere). They too receive the assigned, unique `@move-id`.

10.3. Special Treatment of Text Nodes

The processing of text nodes differs in some respects from regular element nodes. Text nodes are compared by the matcher via so-called *snippet matches*.

- A snippet match consists not only of the pair of text nodes involved, but also mandatorily contains start and length specifications that exactly define which character string (snippet) within the text value matches.
- A single text node can certainly have multiple snippet matches to different partner nodes.

In principle, the algorithm treats these snippet matches very similarly to normal node matches. However, there are some differences. The most important ones are:

1. Steps (c) and (e) (Deletion and Insertion) are executed for *all* snippet matches that do not possess a partner which is a direct child of the corresponding parent counterpart. Each snippet match leads to a separate deletion/insertion. When sorting, snippet matches involving the same node are sorted among themselves according to the corresponding start position within this node.
2. Instead of the `<remove>` and `<insert>` actions, `<update-value>` actions are generated for text nodes. These make it possible to delete or insert only specific parts (snippets) of a text node in a highly granular way, without having to discard the entire text node.

11. Creating An Annotated Diff Result

The primary result of the diff process is an enriched XML document from which both the source and the target document can be completely reconstructed. The nodes of the target version (leading version) are adopted exactly, while all differences from the base version are represented strictly in the form of annotations. To guarantee the structural and semantic integrity of the target version, these annotations are persisted as Processing Instructions (PIs, `<?...?>`). PIs are standard-compliant instructions that are ignored by XML validators and XSLT processors

in their standard behavior. Thus, the resulting document remains structurally valid while simultaneously containing all the information about the changes made. By simply removing the PI annotations, the target version can be extracted without loss of data; reverting to the base version is somewhat more complex, but is also technically possible without loss of data.

11.1. The Oxygen Change Tracking Format as a Basis

The proprietary, yet widely established change tracking format of the *Oxygen XML Editor* in the XML community was adapted as the syntactic basis.

Changes to a document are represented in this format as follows:

- **Insertions:** Newly added nodes are logically enclosed by the markers `<?oxy_insert_start?>` and `<?oxy_insert_end?>`.
- **Deletions:** Since deleted nodes no longer physically exist in the tree, the differ places an `<?oxy_delete?>` marker at the original position. For later reconstruction, the completely removed subtree is serialized as a flat string and stored in the `@content` pseudo-attribute, escaping special characters (e.g., < to `&lt;`).
- **Attribute Changes:** Modifications to attributes require the PI `<?oxy_attributes?>`, which is the direct precessor of the affected element. For each changed attribute, the marker receive a pseudo-attribute of the same name, whose value is a complex "Attribute Change Descriptor" – a serialized `<change>` element that documents the modification type and the original attribute value.

The decision to use this specific format offers significant synergy effects: the format is widely used within the XML community. Other programmes based on this format can be reused. More important, the Oxygen XML Editor is natively able to display these change tracking PIs in its visual Author view without any additional configuration for the end user.

11.2. Semantic Detection of Moves

A particular strength of the XML differ lies in its ability to semantically detect moved content (moves) through various heuristics:

- **ID Matching:** If elements possess identical IDs in both document versions, the differ identifies them as corresponding and marks them, depending on the context, as moved.
- **Content Similarity Matching:** The algorithm can also detect moved nodes based on high textual similarity. If, for example, regular text paragraphs (`<para>`) are converted into list items (`<listitem>` within an `<itemizedlist>`) and the content remains identical or at least very similar, the differ can recog-

nize that the paragraph nodes were merely moved into the newly added list structure nodes.

Since the native Oxygen format does not natively support the representation of such complex structural shifts, it was extended by three conceptual mechanisms:

1. **The @move-id Attribute:** If a move is detected, the `<?oxy_delete?>` marker at the source location receives a unique `@move-id` that points to the target, instead of the `@content` attribute. At the new target location, the `<?oxy_insert_start?>` marker is added by exactly the same `@move-id`.
2. **The @mid Attribute for Nested Modifications:** Since moved content can also be modified in terms of content within the same work step, nested start and end markers are needed. To resolve parser conflicts and overlaps, associated marker pairs are extended by a unique `@mid` attribute for mutual referencing.
3. **Move Placeholders in Deleted Content:** In the event that a comprehensive subtree (e.g., an entire chapter) is deleted, but a subordinate node (e.g., a sub-section) is preserved and moved to another location, the differ inserts a special move placeholder directly into the serialized deletion (`<?oxy_delete?>`) of the main chapter. This placeholder points to the real new position of the moved content via the `@move-id`.

11.3. Application of the Edit Script

The creation of the annotated output file is carried out by applying an edit script. The script is primarily designed so that the sequential execution of all actions contained therein transforms the base version into the target version. However, to generate an annotated version, the script actions must inject corresponding annotation markers in addition to the originally intended tree modifications (such as inserting a corresponding `<?oxy_delete?>` marker instead of simply deleting a node).

This continuous execution poses a significant technical challenge: Since the output of action *N* serves as direct input for action *N+1*, newly inserted annotation markers change the node indices. While markers can generally be easily ignored during index-based node selection, the manipulation of text nodes leads to massive problems. If an action modifies only a partial area of a text node, the inserted marker inevitably splits this single text node into several separate text fragments. The subsequent action, however, expects a contiguous node and consequently fails due to this fragmentation.

To solve this problem, the algorithm implements a dedicated placeholder concept: Before executing the first editing action, all text nodes are encapsulated into temporary proxy elements assigned to a unique namespace. These helper elements act as logical proxies for the original text nodes. An annotation can now split the text within this proxy element without destroying the logical existence

and addressability of the encapsulated node for subsequent actions. After the edit script has been completely processed, these temporary proxy elements are removed again and replaced by their final annotated text content.

12. Results, Future Work, and Conclusion

Currently the XML Differ can correctly generate edit scripts between versions, and also forms the basis for PDF generation illustrating changes.

The changes found are not always optimal. Sometimes elements are marked as deleted and a similar element inserted, and there could in the future be a matcher to detect this. Tables are handled, but column insertions and deletions are not perfect.

A configuration file is needed in order for the XML Differ to be a generally usable tool, but that is not currently implemented.

Performance is adequate for the intended use, and the flexibility and openness of the matcher pipeline seem worth sacrificing speed.

Overall, the XML Differ is useful and and works.

Bibliography

[1] Caumanns, Jörg: *A Fast and Simple Stemming Algorithm*, Free University of Berlin. `https://www.inf.fu-berlin.de/lehre/WS98/digBib/projekt/_stemming.html` (accessed May 2026)

[2] Faasen, Maerijn: *natural-xml-diff: an XML Diffing Library*, Paligo, at XML Prague, 2024. `https://archive.xmlprague.cz/2024/files/xmlprague-2024-proceedings.pdf#page=291` (accessed May 2026)

[3] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesly, 1994.

[4] Salton, G; McGill, M. J.: *Introduction to Modern Information Retrieval,*, Aukland, 1983.

[5] Chawathe, Sudarshan S.; Rajaraman, Anand; Garcia-Molina, Hector; Widom, Jennifer: *Change Detection in Hierarchically Structured Information*, Stanford University, 1996. `http://infolab.stanford.edu/c3/papers/html/tdiff3-8/tdiff3-8.html` (accessed May 2026)

[6] Eugene W. Myers: *An O(ND) Difference Algorithm and Its Variations*, University of Arizona, Tucson, 1986. `https://neil.fraser.name/writing/diff/myers.pdf` (accessed May 2026)

[7] Michael Kay (XSL WG), Saxonica *XPath and XQuery Functions and Operators 3.0*, Stanford University. `https://www.w3.org/TR/xpath-functions-30/`

Jiří Kosek (ed.)

XML Prague 2026
Conference Proceedings

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and AH Formatter.

1st edition

Prague 2026

ISBN 978-80-907787-4-0 (pdf)
ISBN 978-80-907787-5-7 (ePub)