# Markup UK

## 2025 Proceedings

A Conference about XML and Other Markup Technologies

**Silver**

A Data Usability Company
ANTENNA HOUSE

SAXONICA.COM
XSLT AND XQUERY PROCESSING

**Bronze**

## Markup UK

A Conference about XML and Other Markup Technologies
https://markupuk.org/

Markup UK Conferences Limited
is a limited company registered in England and Wales.
Company registration number: 11623628
Registered address: 24 Trimworth Road, Folkestone, CT19 4EL, UK
VAT Registration Number: 316 5241 25

## Organisation Committee

Geert Bormans
Ari Nordström
Andrew Sales
Rebecca Shoob

## Programme Committee

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

## Thank You

Evolved Binary
le-tex Publishing Services
Saxonica
OxygenXML
Ilmari Koria
Wissam Asfahani
...and our long-suffering partners

## Sister Conferences

Balisage
The Markup Conference

xmlprague

XML
summer school

Declarative
Amsterdam

## Markup UK 2025 Proceedings

by Rebecca Bamford, Achim Berndzen, Francis Cave, Charafeddine Cheraa, John Cummins, Francis Denton, Tony Graham, Gerrit Imsieke, Michael H Kay, Martin Kraetke, Astrea Kumaradas, Deborah A Lapeyre, David Maus, Ari Nordström, Steven Pemberton, Liam Quin, Adam Retter, Andrew Sales, Erik Siegel, Amber Smiley, Sheila Thomson, Norman Tovery-Walsh, B. Tommie Usdin and Christine Windeln.

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

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

# Markup UK

# Modular ixml

Steven Pemberton, CWI, Amsterdam

Most current ixml grammars are small. However there are examples of large grammars, and it is likely that in the future more large grammars will emerge as ixml usage increases.

To make large grammars more manageable, and to enable reuse, it would be useful to have a way to modularise them.

One of the requirements of modularisation for reuse in any notation is to have a method of specifying the contractual interface, such that it is possible for the producers of the modules to change their internal structure without breaking any existing usage of the module.

This paper describes a proposal for an ixml preprocessor that permits an ixml grammar to invoke other modules of ixml grammars, specifying their linkage. This involves the renaming of rules with name clashes in the modules, using ixml renaming, resulting in a single ixml grammar with no rule-name clashes, and so that the resultant XML serialisations remain the same. The invoking grammar remains unchanged.

There is no change to the syntax or semantics of ixml proper.

## 1. Contents

## 2. Introduction

Invisible XML, ixml for short [ixml], is a notation and process that uses context-free grammars to describe the format of textual documents, allowing documents to be parsed into an abstract parse-tree, which can be processed in various ways, but principally serialised into an XML document, thus making the implicit structure of the textual document explicit in the XML.

While most current ixml grammars are small (the grammar for ixml itself for example is around 70 lines), it can be envisaged that in the future large grammars will emerge containing subparts that are authored by different people. As an example, there is an ixml grammar for XPath 4 at around 350 lines [jwl] which could be used by grammars for languages that use XPath 4.

In [vdb], van den Brand *et al.* note the advantage of context-free generalised parsing, which is used by ixml, over other restricted forms:

> *"the class of context-free grammars is closed under union, in contrast with all proper subclasses of context-free grammars. [...] The compositionality of context-free grammars opens up the possibility of developing modular syntax definition formalisms. Modularity in programming languages and other formalisms is one of the key beneficial software engineering concepts."*

What this is saying, is that if you have, for instance, two LL1 grammars and merge them, the result may not still be LL1, but if you merge two general context-free grammars, the result will still be context-free, and this is one of the advantages of context-free generalised parsing, that you can modularise them.

## 3. Requirements

The main problem with merging two independent context-free grammars is that grammars have no inherent scoping, and grammar rules in different component grammars may have the same name, thus causing a clash. Modularisation has then to be designed so as to prevent these name clashes. While this is the central functional design need for modularisation, a number of other requirements and desiderata were formulated for the design:

- It should be designed as a preprocessor that takes modules and an invoking grammar, and produces a single ixml grammar as output; in this way it will work with all existing ixml processors without change.
- Consequently, there should be no change required to ixml proper: just use the existing syntax and semantics.
- Modules should be able to invoke other modules.
- Modules should have a visible contract of use, on both the producer's as the user's side, so that it is obvious what each module uses and shares, and that if there are different implementations of a particular module they can be swapped in and out.
- The internals of a module should be protected, so that a module owner can change the implementation of a module, as long as the interface contract is maintained.
- It should be possible to independently check modules for completeness and consistency, so that modules can be checked before they are combined.
- Although modules may be transformed to prevent name clashes, no changes should be made to the invoking grammar, so that error messages are in the user's terms, and not using renamed terms.
- Despite rules being renamed, the resultant serialisation should not change.
- Modularisation should be kept as simple and easy-to-use as possible while meeting the requirements.

## 4. Naming and renaming

The modularisation proposed here uses a new feature of ixml: *renaming*, a feature agreed by the working group, but not yet part of the official specification; it is specified in the current working draft [wd] and already present in several implementations. It allows you to specify for a rule a different name than the default to be used on serialisation.

To illustrate: an ixml rule has a name. Up to now in ixml, this specifies a name both for the allowable input syntax, as for the name used in the output serialisation for that rule. If two input forms have different syntaxes, it is therefore necessary to give them different names, even if the intention is to have the same output serialisation.

For instance, consider a grammar that accepts both `31/12/1999` and `31 December 1999` forms of dates:

```
    date: numeric; textual.
-numeric: day, -"/", month, -"/", year.
-textual: day, -" "+, tmonth, -" "+, year.
     day: d, d?.
   month: d, d?.
    year: d, d, d, d.
  tmonth: -"January",  +"1";
          -"February", +"2";
          ...
          -"December", +"12".
      -d: ["0"-"9"].
```

What you will see is that the serialisation of these are nearly identical, except that while `31/12/1999` produces

```
<date>
   <day>31</day>
   <month>12</month>
   <year>1999</year>
</date>
```

`31 December 1999` produces

```
<date>
   <day>31</day>
   <tmonth>12</tmonth>
   <year>1999</year>
</date>
```

where the difference is because it is produced from a different input syntax. Using renaming, you can specify that both have the same serialised name:

```
tmonth > month:
   -"January",  +"1";
   -"February", +"2";
   ...
   -"December", +"12".
```

This says that while `tmonth` is the name used in the grammar, and represents the textual form of a month in the input, it should be serialised as `month`, thus in this case making the two date serialisations identical.

Incidentally, since the allowable ixml names are not exactly the same set as the allowable XML names, you can also specify the renaming as a string. For instance since ixml names may not end with a dot, but XML names may, you can write:

```
abc > "abc.": ...
```

The syntax of the start of a rule like this is called a `naming`, and can consist either of a `name`, as currently in ixml, or a `renaming`, which consists of a name, a greater than, and an *alias*, which can either be a name or a string.

Also in passing, it is worth noting that this has consequences for round-tripping, as presented in [rt], since this introduces a roundtripping ambiguity. Because an output form such as

```
<date>
   <day>31</day>
   <month>12</month>
   <year>1999</year>
</date>
```

can have been produced by two different input syntaxes, the roundtripping process has to choose one of them. Where necessary this can be overcome with a technique such as:

```
tmonth > month:
   style,
   (-"January",  +"1";
    -"February", +"2";
    ...
    -"December", +"12").
@style: +"text".
```

which would produce for the `31 December 1999` style of input

```
<date>
   <day>31</day>
   <month style='text'>12</month>
   <year>1999</year>
</date>
```

which can be uniquely round-tripped.

With this background explained, we can now proceed to the design of modularisation.

## 5. The Structure of a Module

A module consists of an otherwise normal ixml grammar, preceded by specifications of rules used from other modules and what is shared for use from this module.

A specification of what to use from another module lists the rules needed from each module it uses. Such a specification should be recognisable as different from an ixml rule.

The character to signal such a specification has been chosen as "+", though any character that doesn't start the first ixml rule in a grammar could have been used in the design; ixml rules can start with namestart characters, "-", "^" (and "@" but it is not possible to start the first rule of a grammar with that character):

```
+uses css from css.ixml
```

and

```
+uses iri, url, uri, urn from uri.ixml
```

This specifies which module to use, and which rules from that module are intended to be used.

It is possible to combine them

```
+uses css from css.ixml; iri, url, uri, urn from uri.ixml
```

The specification of what is allowable to be used from a module is similar:

```
+shares iri, url, uri, urn
```

There are two main choices for a grammar for these. The first literally recognises the structure as it is specified above:

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

where `s` is the regular ixml rule for optional whitespace, `rs` for required whitespace, `name` the rule for a rule name, `ixml` the rule for an ixml grammar, and `iri`, not defined here, representing an internationalised URI [iri], allowing the use of grammars from external sources, such as:

```
+uses iri from https://example.com/ixml/modules/iri.ixml
```

For a specification like

```
+uses css from css.ixml; iri, url, uri, urn from uri.ixml
```

this produces a resulting structure like

```
<uses>
   <from source='css.ixml'>
      <share name='css'/>
   </from>
   <from source='iri.ixml'>
      <share name='iri'/>
      <share name='url'/>
      <share name='uri'/>
      <share name='urn'/>
   </from>
</uses>
```

Alternatively, the grammar could look like:

```
   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.
```

where the resulting structure then looks like:

```
<uses from='css.ixml'>
   <share name='css'/>
</uses>
<uses from='uri.ixml'>
   <share name='iri'/>
   <share name='url'/>
   <share name='uri'/>
   <share name='urn'/>
</uses>
```

The advantage of the latter version is that processing is slightly easier, since shallower, with a slight disadvantage with respect to round-tripping, since the two forms

```
+uses css from css.ixml; iri, url, uri, urn from uri.ixml
```

and

```
+uses css from css.ixml
+uses iri, url, uri, urn from uri.ixml
```

are no longer distinguishable on roundtripping, since they produce the same serialisation.

## 6. Semantics

There are some semantic requirements:

- all modules used must exist;
- a module must share the rules mentioned to be used from that module;
- all names in `uses` and `shares` specifications in a module must be unique;
- a module must not define a rule for any name that it `uses`;
- a module must define rules for all names it `shares`.

Modules are allowed to invoke each other: consider a programming language where declarations can include procedures, and procedures can include declarations, then the module for procedures would have:

```
+uses declaration from declaration.ixml
+shares procedure
```

and the module for declarations would have:

```
+uses procedure from procedure.ixml
+shares declaration
```

This illustrates that a `uses` specification is different from, for instance, a `#include` statement in C preprocessing, since `uses` only ensures that the module will be present in the final grammar.

Note that a module can only share rules it defines; it is not permitted to share a rule from a different module like this:

```
+uses x, y from z.ixml
+shares x
```

So, having defined what a module looks like, we can now use it to define itself:

```
+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.
```

## 7. Processing

The set of the invoking module and all invoked modules is collected, including modules in turn invoked by those modules. These modules are going to be concatenated, but any name clashes are resolved first.

If any two invoked modules contain the definition of a rule of the same name, one of the rules is renamed:

- If either of the pair is a rule that is not used in any other of the set of modules (whether shared or not), then that rule is renamed.
- If both are used in other modules, then if one of the pair is a rule defined by the original invoking module, the other is renamed; otherwise either may be renamed.

A rule is renamed by generating a new unique name, different from all other rule names in the set of modules:

- If the rule is defined with a `naming` (i.e. it has a name and an alias), the rule is redefined with a `naming` consisting of the new unique name and the existing alias.
- If the rule is defined using just a name (i.e. without an alias), the rule is redefined with a `naming` formed of the new unique name as name, and the old name as alias.

All applications of the old name in the module grammar, and any of the other modules that use that rule are replaced with the new name.

Once all naming conflicts are resolved, all invoked modules are appended to the invoking module, with the `uses` and `shares` specifications removed.

What these rules ensure is that:

- there are no name clashes in the resulting grammar;
- the original invoking grammar is not changed in any way, so that error messages about that grammar are given in terms the author expects;
- changes within a module are preferred over changes that extend to other modules;
- any resulting serialisation remains the same.

## 8. Example

As a simple example, imagine a language of identity statements of the style

```
total=price+tax+shipping
tax=price×10÷100
shipping=5
```

expressed by this grammar that uses the definition of `expr` from another module:

```
+uses expr from expr.ixml
    data: identity+.
identity: id, -"=", expr, -#a.
      id: [L]+.
```

The only problem is that the `expr` module has a clashing rule for `id`:

```
+shares expr
expr: id++op.
  id: [L; Nd]+.
  op: ["+-×÷"].
```

Since the invoking grammar never gets changed, the rule in the module gets renamed, resulting in the following complete grammar:

```
    data: identity+.
identity: id, -"=", expr, -#a.
      id: [L]+.

    expr: id_++op.
  id_>id: [L; Nd]+.
      op: ["+-×÷"].
```

If the module's rule for `id` had instead been a *renaming*, it could have looked like this:

```
id>ident: [L; Nd]+.
```

and the renaming would have ended up as:

```
id_>ident: [L; Nd]+.
```

## 9. Example

Making the example slightly more complex, with rules like

```
result[1]=a1+b1+c1
result[2]=a2+b2+c2
```

using this grammar:

```
+uses expr from expr.ixml; identity from id.ixml
rules: rule+.
 rule: identity, -"=", expr, -#a.
```

Module `expr.ixml`

```
+shares expr
   expr: operand++op.
operand: id; number.
     id: [L], [L; Nd]*.
     op: ["+-×÷"].
 number: ["0"-"9"]+.
```

Module `identity.ixml` has a clash with both `id` and `number` from `expr.ixml`:

```
+shares identity
identity: id; id, -"[", number, -"]".
      id: [L]+.
  number: digits, (".", digits)?.
 -digits: [Nd]+.
```

The invoking grammar never changes:

```
rules: rule+.
 rule: identity, -"=", expr.
```

In module `expr.ixml` nothing needs changing

```
   expr: operand++op.
operand: id; number.
     id: [L], [L; Nd]*.
     op: ["+-×÷"].
 number: ["0"-"9"]+.
```

In `identity.ixml` both `id` and `number` are renamed:

```
      identity: id_; id_, -"[", number_, -"]".
        id_>id: -"@", [L]+.
number_>number: digits, ".", digits.
       -digits: [Nd]+.
```

The rules allow either or both to be renamed in `expr.ixml` instead.

## 10. Example

The invoking grammar:

```
+uses id from ident.ixml; expr from expr.ixml
rules: rule+.
 rule: id, -"=", expr.
```

Module `ident.ixml`

```
+shares id
id: [L]+.
```

Module `expr.ixml`

```
+uses id, number from id.ixml
+shares expr
   expr: operand++op.
operand: id; number.
     op: ["+-×÷"].
```

Module id.ixml

```
+shares id, number
    id: [L], [L; Nd]*.
number: [Nd]+.
```

Here there are two rules called `id` both shared and used by two different modules.

The invoking grammar is never changed:

```
rules: rule+.
 rule: id, -"=", expr.
```

and since the `id` rule is used from module `ident.ixml`, the rule may not be renamed there:

```
id: [L]+.
```

This means that the `id` rule in module `id.ixml` has to be renamed:

```
id_>id: [L], [L; Nd]*.
```

```
number: [Nd]+.
```

and in module `expr.ixml` that uses it

```
   expr: operand++op.
operand: id_; number.
     op: ["+-×÷"].
```

## 11. A Larger Example

Imagine you were defining a textual format for XForms [xf]:

```
Example XForm
style xform.css

model M
   instance data data.xml
   submission save put:data.xml replace:none

input name "What is your name?"
submit "OK"
```

This is going to need definitions for CSS, URIs, XPath, and a lot more. Furthermore, it would be worth modularising it into several parts that are effectively independent, reflecting the model-view-controller aspect of XForms: the model, the content, and actions. This might result in a grammar like this (this is not a complete example).

The top-level:

```
+uses form from form.ixml
+uses content from content.ixml

xform>html: h, form, content.
  @h>xmlns: +"http://www.w3.org/1999/xhtml".
```

Module `form.ixml`:

```
+shares form
+uses css from css.ixml;
model from model.ixml;
iri from iri.ixml;
s from xforms-basics.ixml

     form>head: title, styling?, model*.
         title: ~[" "; #a], ~[#a]+, -#a.
      -styling: -"style", s, (style; stylelink).
stylelink>link: csstype, cssrel, href.
```

```
                style: csstype, css.
        @csstype>type: +"text/css".
          @cssrel>rel: +"stylesheet".
                @href: -iri, s.
```

Module `model.ixml`:

```
       +shares model
       +uses s, ref, xf from xforms-basics.ixml;
       id, name from xml.ixml;
       Action from action.ixml;
       iri from iri.ixml

            model: -"model", s, id, s, xf, -#a,
                      s, (instance; bind; submission; Action)+.

         instance: -"instance", s, id, s, resource, s.
        @resource: -iri.

             bind: "bind", s, (id, s)?, ref, s, property*.
         property: type {; readonly; relevant; required; etc}.
             type: "type:", name, s.

       submission: -"submission", s, id, s,
                      (method, -":", resource, s)?, replace?.
          @method: "get"; "put".
         @replace: -"replace:", name, s.
       {etc}
```

Module `content.ixml`:

```
       +shares content
       +uses IDREF from xml.ixml;
       xf, ref, string, s from xforms-basics.ixml

       content>body: group.

              group: xf, control*.
           -control: input; submit {more}.

              input: -"input", s, ref, label.
              label: string.

             submit: -"submit", s, subid?, label?.
  @subid>submission: -"submission:", IDREF, s.
```

and so on. Giving output like:

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

```
            <head>
               <title>Example XForm</title>
               <link type='text/css' rel='stylesheet'                ↩
href='xform.css'/>
               <model id='M' xmlns='http://www.w3.org/2002/xforms'>
          <instance id='data' resource='data.xml'/>
          <submission id='save' method='put' resource='data.xml'     ↩
replace='none'/>
               </model>
            </head>
            <body>
               <group xmlns='http://www.w3.org/2002/xforms'>
          <input ref='name'>
             <label>What is your name?</label>
          </input>
          <submit>
             <label>OK</label>
          </submit>
               </group>
            </body>
         </html>
```

## 12. Other Possible Approaches

This proposal introduces a modularisation that allows modules to be combined while only requiring minimal changes to avoid name-clashes. Other approaches would include introducing scoping into ixml, (but this would just move responsibility for renaming to the implementations), or renaming all rules, for instance by in some way including the module 'name' (=source) in each rule name. This would involve more changes during processing, but might result in less analysis of rule names.

## 13. Conclusion

What this proposal has shown is that you can introduce modularisation in ixml by imitating scoping in a simple and direct manner, allowing a pre-processor to produce a complete ixml grammar that produces an identical serialisation of the parsed input, without having to change the syntax or semantics of ixml proper.

## References

[iri] M. Duerst and M. Suignard. *RFC 3987: Internationalized Resource Identifiers (IRIs)*. IETF. 2005. https://datatracker.ietf.org/doc/html/rfc3987 .

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

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

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

[vdb] M.G.J. van den Brand et al.. *Disambiguation Filters for Scannerless Generalized LR Parsers*. Compiler Construction. 2002. 143–158. https://doi.org/10.1007/3-540-45937-5_12 . https://cwi.nl/~jurgenv/papers/CC-2002.pdf .

[wd] Steven Pemberton. *Invisible XML Specification - Community Group Editorial Draft*. Invisible XML Community Group. 2025. https://invisiblexml.org/current/ .

[xf] Erik Bruchez et al.. *XForms 2.0*. W3C. 2025. https://www.w3.org/community/xformsusers/wiki/XForms_2.0 .

Markup UK

# From iXML to XSpec

Sheila Thomson

In the autumn of 2024, I attended the XML Summer School and persuaded (it wasn't difficult) Stephen Pemberton to tack a session on iXML onto the end of his course on XForms. On the bus home, I made a start on my first iXML grammar. Within a week I had encountered a character encoding mystery and whitespace challenges, lost a day naming things and was working on repurposing XSpec for testing iXML grammars.

This paper: outlines my learning journey, as a case study; highlights existing methods for tesing iXML; and summarises how I adapted XSpec for testing iXML.

## 1. Introduction

In the autumn of 2024, I attended the XML Summer School and persuaded (it wasn't difficult) Stephen Pemberton to tack a session on iXML onto the end of his course on XForms. On the bus home, I made a start on my first iXML grammar. Within a week I had encountered a character encoding mystery and whitespace challenges, lost a day naming things and was working on repurposing XSpec for testing iXML grammars.

This paper:

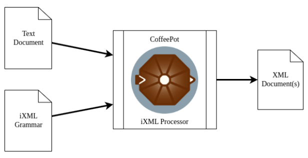
- outlines my learning journey as a case study
- highlights existing methods for testing iXML
- summarises how I adapted XSpec for testing iXML

### 1.1. What is iXML?

The specification for Invisible XML (iXML) describes it as "a method for treating non-XML documents as if they were XML"[IXML]. On Invisible XML [http://invisiblexml.org/], where the spec is hosted, it expands that definition slightly: "Invisible XML is a language for describing the implicit structure of data, and a set of technologies for making that structure explicit as XML markup."[IMCG]

In practice, this means that if a document or dataset has a regular enough structure that you can write down the rules to parse it (an iXML grammar), then you should be able to use an iXML processor (such as CoffeePot) to apply the grammar to the document or dataset in order to convert it to XML (see Figure 1 [22]).

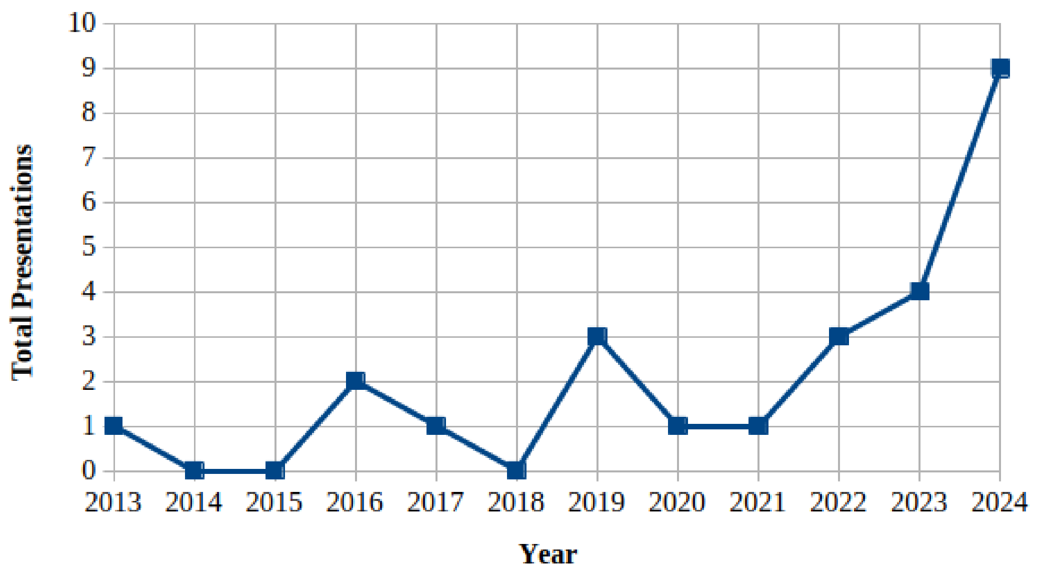
Figure 1. High-level overview of an iXML conversion



### 1.2. Why iXML?

People have been talking about iXML for a few years now and it's become a re-occurring topic at markup conferences (see Figure 2 [23][1]). In my case, I was looking for a new personal project and curious to find out whether iXML would live up to the hype.

**Figure 2. Frequency of Presentations on iXML**



Also, I suspect that I may be able to use iXML to simplify an existing pipeline that converts family tree data from GEDCOM to XML. GEDCOM itself is a standard, with a grammar, and although it can be used to serialise data as XML or text, it seems that the text option is most common; it's certainly the only option for exporting from Ancestry.com [ancestry.com], which is where I maintain my family trees.

For an in-depth consideration of why you, or a client, might want to use iXML, see past papers on the topic, particularly Steven Pemberton's original proposal for Invisible XML, that he presented at Balisage in 2013[PEM2013].

## 2. Learning iXML

Steven Pemberton often runs tutorials for learning iXML at markup conferences and, although it wasn't on the schedule, he very kindly tacked one onto the end of a course I was attending on XForms, during the 2024 XML Summer School [https://xmlsummerschool.org/], in Oxford.

If you're unable to attend one of Steven's tutorials, Norm Tovey-Walsh has created an online tutorial *Writing Invisible XML grammars* [https://www.xml.com/articles/2022/03/28/writing-invisible-xml-grammars/]. The *Invisible XML Specification* is published at https://invisiblexml.org/1.0/ and I recommend paying particular attention to the *Complete Grammar* section so you're aware of the pre-defined rules; it's also a great example of what an iXML grammar looks like and immediately after it is an example of the XML that would be generated if you applied the iXML grammar to itself. The spec is developed and maintained by the W3C Invisible Markup Community Group [https://www.w3.org/community/ixml/], who have a email discussion group [https://lists.w3.org/Archives/Public/public-ixml/] that anyone can message and a public archive that you can search (in case your question has been

[1]At Balisage, Declarative Amsterdam, Markup UK, XML London and XML Prague. For the full list, see Appendix B [34]

raised and discussed before). There is a list of grammars on https://invisiblexml.org/ but in general I've struggled to find publicly available iXML grammars.

For my own first attempt at writing an iXML grammar, I chose the DTD document type (DOCTYPE) declaration "string" as the text I wanted to parse. Although DTDs have technically been superseded by more modern schema languages, they're still fairly commonplace. The DOCTYPE declaration can be used to associate an XML document with an external Document Type Definition (DTD). I chose external because it's much simpler than internal and the use-cases I had in mind all related to external definitions.

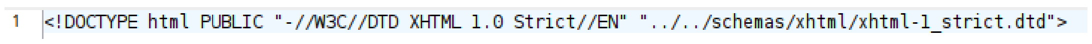
When working with XML documents, sometimes it's useful or necessary to know what the exact content model is. In human mode, I can open the document and read the schemas association information; if that's not there, we're into a whole other problem that's out-of-scope for this particular use case. Programmatically, it's not so easy, as that information is discarded when a DOM object is created.

Another reason for choosing the DOCTYPE declaration was that the rules for writing one are explicitly defined in the specification for XML[XML1-0] so I wouldn't have the additional overhead of trying to infer them from sample documents.

Once I started drafting my iXML grammar, I soon wanted to test it out and "eyeball" what was coming out. We hadn't been running iXML transformations locally on our laptops during the tutorial (because it was unplanned) so I wasn't already set up for it. Happily, Google led me to CoffeePot, an Invisible XML processor, developed and maintained by Norm Tovey-Walsh[CPOT]. It's distributed as a jar file, can easily be run from a command-line and has a useful manual. For a list of alternative implementations, see invisblexml.org.

Next, I needed a sample input document. This is when I discovered that it wasn't possible to use iXML itself to find and isolate the DOCTYPE declaration within the XML document; unlike a regular expression, an iXML grammar must match the entirety of the input document. At this point, as I was simply looking for sample input to experiment with, I should have just manually copy and pasted the DOCTYPE declaration string into a file of it's own. However, I had it stuck in my head that my real-world use-case was to programmatically parse the string and convert it to XML. So I created an ANT macro to load the XML document as text and use a regular expression to find and extract the DOCTYPE declaration. I could have made my regular expression more complex in order to identify the component parts of the DOCTYPE declaration but that would have defeated the point of the learning exercise. Also, one of the advantages of an iXML grammar over a regular expression is readability. Understanding even a slightly complex regular expression written by someone else - or by myself a while ago - can be challenging whereas an iXML grammar is commonly expressed as named components so it's easier to identify the logical "building blocks" and how they relate to each other.

**Figure 3. Single-line text file containing a DOCTYPE declaration for XHTML**



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "../../schemas/xhtml/xhtml-1_strict.dtd">
```

Figure 3 [#doctype-declaration-example] shows an example of a DOCTYPE declaration string after it's been extracted from an XHTML document and saved separately in a text file.

Now I was ready to test.

## 3. Testing an iXML grammar

For general reasons to test, see Sandro Cirulli's 2017 paper introducing *XSpec v0.5.0*[CIR2017] and *Stretching XPath: Three Testing Tales: Beyond Primary Use Cases of Certain XML Functions and Standards* by Amanda Galtman[GAL2024].

### 3.1. Eyeballing the output

An informal term for literally just looking at the output and manually scanning for errors. Unless you're a hardcore practitioner of Test Driven Development[TDD], it's likely that you've done this and it's certainly what I was doing when I first started experimenting with iXML. It's low risk while you're learning something new and the inputs and outputs are small and simple - and the software isn't in production - but as complexity increases, manual checks usually become more time-consuming and risky and the odds are that you will start missing mistakes. Eventually you're going to need to put something more robust in place.

### 3.2. Schema validation

If there's a schema for your target XML then you may be able to validate your output against it. In some cases, even though a schema is available this type of testing may not be practical, for example, if your iXML creates content to be inserted into a larger document and the schema is designed to only validate in the context of that larger document.

Schema validation can alert you to errors such as a misspelt element or attribute name, a missing wrapper element or elements in the wrong order. This is especially pertinent to testing an iXML grammar as this is where many of (sometimes all) the element and attribute names are written.

However, most schemas check the markup but not the content that is being marked up so, for example, if a word or URL is unintentionally changed during the conversion, a schema is unlikely to detect it.

A schema also won't catch content that is unintentionally lost during the conversion unless the lost content is mandatory.

### 3.3. Diff

Diffing is similar to eyeballing the content but the comparison is executed programmatically, so if you want to use it as a testing method, you need to provide the diff tool with a copy of the output expected (a control document) as well as the content to check.

This technique is especially useful if you don't have a schema to validate against and also for spotting lost or unintentionally changed content. However, if the control document is large, it can be very time-consuming to create and maintain.

### 3.4. XSpec

XSpec is a popular testing framework, originally created in 2008[CIR2017] for testing XSLT but it has since been extended to also support XQuery and Schematron.[WHATXSP] My experiment to use XSpec for testing iXML extended the process for testing XSLT so the scope of this section is limited to the XSpec process for XSLT. However, Amanda Galtman has published an article on how to test iXML using XQuery in BaseX with Markup Blitz. [GALMED]

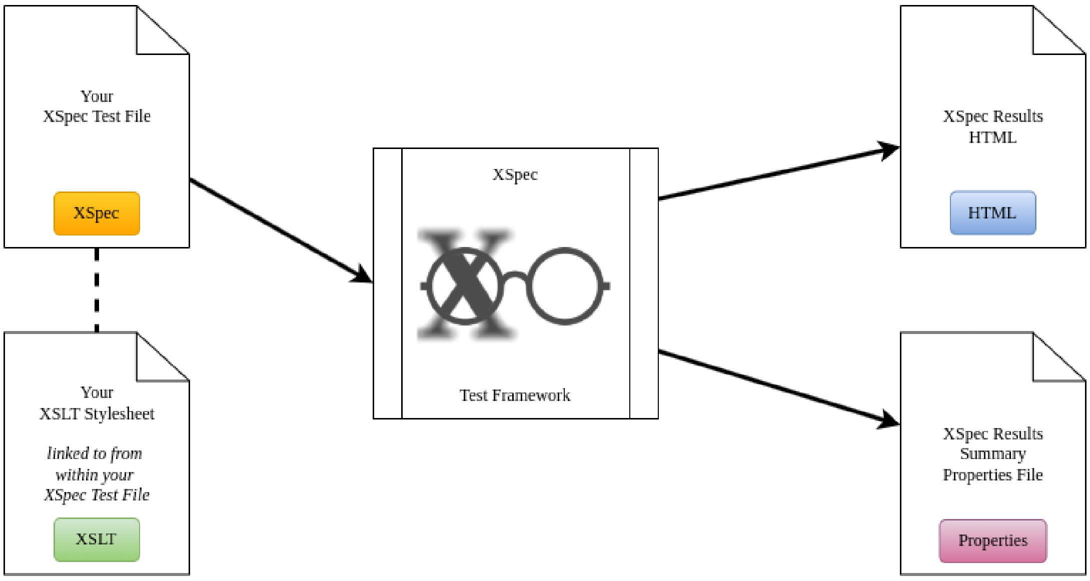
#### 3.4.1. Using XSpec to test XSLT

Figure 4 [#xspec-xml-for-xslt], shows the XML for a very simple XSpec file (`jumping.xspec`) to test an XSLT file named `jumping.xsl`. The input data is provided inline, wrapped within an `x:context` element (line 10). This example contains three tests (lines 12-14).

Figure 4. Simple XSpec test file

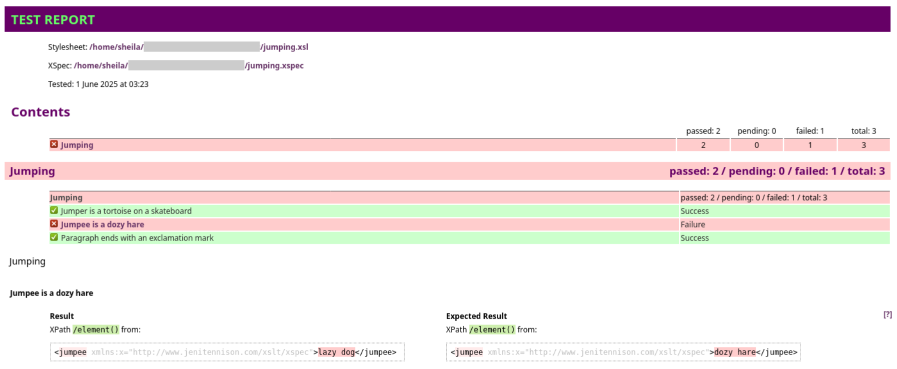
XSpec takes the test file as an input and uses `/x:description/@stylesheet` to locate the XSLT file (see Figure 5 [26]). When it's finished evaluating the tests, it outputs two result documents: one containing a detailed breakdown of the results and showing any errors found (HTML) and the other a simple ANT properties file containing the overall result (pass/fail).

Figure 5. Inputs and outputs when testing XSLT with XSpec (high-level)



For the purposes of this example, the XSLT hasn't been updated to change "lazy dog" to "dozy hare" so the second test (line 13) fails. See Figure 6 [xspec-xslt-test-report] for an screenshot of the HTML results file. Notice that for the failed test, the report includes a diff visualisation of the actual result returned by the XSLT (on the left) and the "Expected Result" that was specified on line 13 of the XSpec test file.

Figure 6. XSpec results (HTML)



TEST REPORT

Stylesheet: /home/sheila/[redacted]/jumping.xsl

XSpec: /home/sheila/[redacted]/jumping.xspec

Tested: 1 June 2025 at 03:23

Contents

| | passed: 2 | pending: 0 | failed: 1 | total: 3 |
|---|---|---|---|---|
| ✗ Jumping | 2 | 0 | 1 | 3 |

Jumping — passed: 2 / pending: 0 / failed: 1 / total: 3

| Jumping | passed: 2 / pending: 0 / failed: 1 / total: 3 |
|---|---|
| ✓ Jumper is a tortoise on a skateboard | Success |
| ✗ Jumpee is a dozy hare | Failure |
| ✓ Paragraph ends with an exclamation mark | Success |

Jumping

**Jumpee is a dozy hare**

**Result**
XPath `/element()` from:

```
<jumpee xmlns:x="http://www.jenitennison.com/xslt/xspec">lazy dog</jumpee>
```

**Expected Result** [?]
XPath `/element()` from:

```
<jumpee xmlns:x="http://www.jenitennison.com/xslt/xspec">dozy hare</jumpee>
```

One of the key advantages of XSpec over other diff tools is the option to use XPath to target and test sub-sections of content. For more in-depth tutorials, see the *Getting Started* section on the official XSpec wiki.[GETST]

### 3.4.2. Using XSpec to test iXML

One of the maintainers of XSpec, Amanda Galtman, has identified a relatively simple way that this can be done using the `invisible-xml` function in XPath 4.0[GALMED][XPATH4]. Unfortunately, the specification for XPath 4.0 is still in development and not yet widely supported. However, Amanda's solution can already be used if you write your

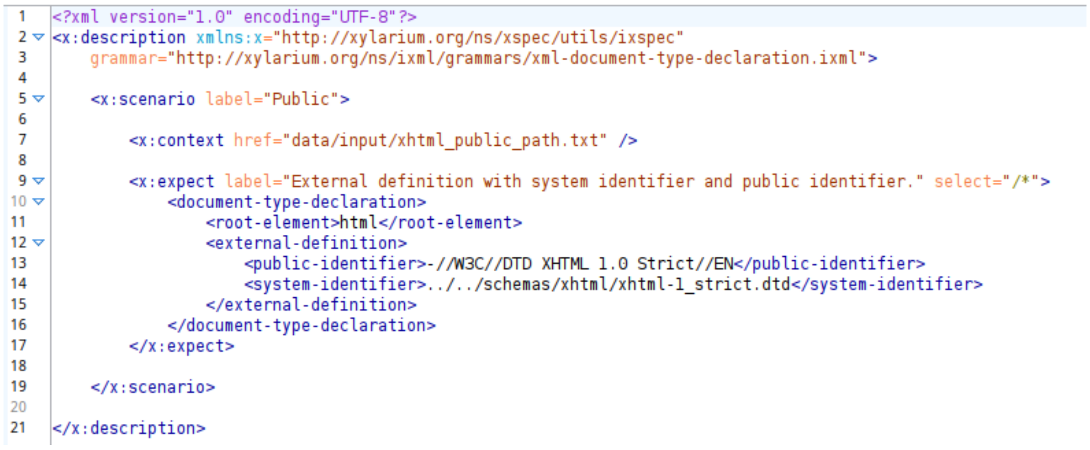
XSpec tests as XQuery and run them in BaseX with the Markup Blitz library. Once the `invisible-xml` function is supported in an XSLT processor, it should also be possible to use it in XSpec tests written for XSLT.

When I started thinking about how XSpec's XSLT process might be adapted for testing iXML, the main differences that I identified were:

1. the context is text, not XML
2. the code being tested is an iXML grammar, not an XSLT stylesheet

I wanted to make as few changes as possible to the way that the XSpec test file would be written[WRITEXSPEC] and realised that this would be possible if I could apply the iXML transformation to the context before the expectations were evaluated. So that the XSpec would be able to easily differentiate a test file written for iXML from one written for XSLT, XQuery or Schematron, I added a new attribute to the root element of the test file, named `grammar`, the value of which is expected to be a path to a file containing the iXML grammar being tested.

**Figure 7. An example of an XSpec test file for iXML**

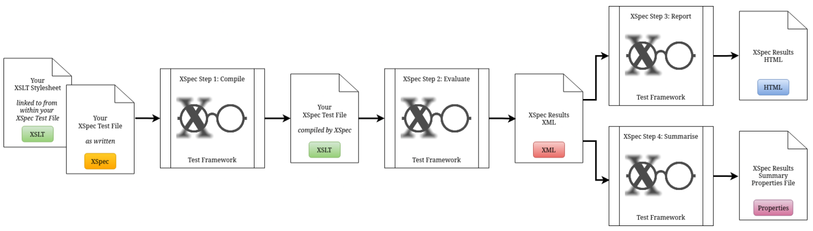


```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <x:description xmlns:x="http://xylarium.org/ns/xspec/utils/ixspec"
3       grammar="http://xylarium.org/ns/ixml/grammars/xml-document-type-declaration.ixml">
4
5       <x:scenario label="Public">
6
7           <x:context href="data/input/xhtml_public_path.txt" />
8
9           <x:expect label="External definition with system identifier and public identifier." select="/*">
10              <document-type-declaration>
11                  <root-element>html</root-element>
12                  <external-definition>
13                      <public-identifier>-//W3C//DTD XHTML 1.0 Strict//EN</public-identifier>
14                      <system-identifier>../../schemas/xhtml/xhtml-1_strict.dtd</system-identifier>
15                  </external-definition>
16              </document-type-declaration>
17          </x:expect>
18
19      </x:scenario>
20
21  </x:description>
```

In Figure 7 [27], the iXML grammar is specified on line 3 and the text to be converted to XML is stored in a separate file and specified using the `href` attribute on `x:context` (line 7). This is a standard option in XSpec that's very useful if you're re-using the same text in multiple scenarios. The contents of `xhtml_public_path.txt` can be seen in Figure 3 [24]. The expected result is specified, as per usual, via the `x:expect` element (lines 9-16). Similar though it is, this test file is not a valid XSpec file so I also changed the namespace URI (line 2). During the iXML pre-processing step, when a valid XSpec file is created, the namespace URI is changed back to `http://www.jenitennison.com/xslt/xspec`.

Under the hood of the XSpec testing process, there are four main steps: compile, evaluate, report and summarise.

**Figure 8. Steps in the XSpec process, with inputs and outputs**

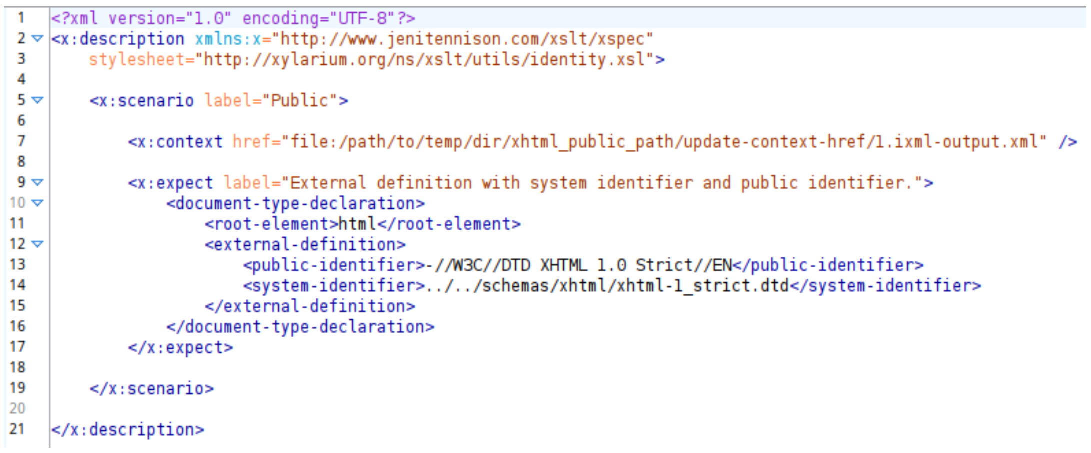


I decided to add the iXML transformation as an extra step, immediately before Compile; it's not unprecedented, as XSpec already includes a pre-processing step when testing Schematron. I implemented this step using XProc, mostly so that I could take advantage of its support for iXML (`p:ixml`[2])[XPROCIXML] but also, if I'm honest, in part because it was a personal project and I enjoy working with XProc. This custom XProc step:

1. changes the namespace URI from `http://xylarium.org/ns/xspec/utils/ixspec` to `http://www.jenitennison.com/xslt/xspec`
2. replaces `/x:description/@grammar` with `/x:description/@stylesheet` which references an XSLT identity transform stylesheet
3. uses `p:viewport` to iterate over all the `x:context` elements and, for each:
   a. uses `p:ixml` to transform the input (`x:context/@href`) into XML using the grammar (`/x:description/@grammar`) and stores the result as a temporary file
   b. changes `x:context/@href` to reference the temporary file containing the result of the iXML transformation
4. uses `resolve-uri()` and `p:xslt` to change any `x:expect/@href` URIs that are relative to absolute.

Figure 9 [28], shows what the output of this step would be if it were applied to the example XSpec file shown in Figure 7 [27].

**Figure 9. XSpec test file for iXML after pre-processing**



```xml
<?xml version="1.0" encoding="UTF-8"?>
<x:description xmlns:x="http://www.jenitennison.com/xslt/xspec"
    stylesheet="http://xylarium.org/ns/xslt/utils/identity.xsl">

    <x:scenario label="Public">

        <x:context href="file:/path/to/temp/dir/xhtml_public_path/update-context-href/1.ixml-output.xml" />

        <x:expect label="External definition with system identifier and public identifier.">
            <document-type-declaration>
                <root-element>html</root-element>
                <external-definition>
                    <public-identifier>-//W3C//DTD XHTML 1.0 Strict//EN</public-identifier>
                    <system-identifier>../../schemas/xhtml/xhtml-1_strict.dtd</system-identifier>
                </external-definition>
            </document-type-declaration>
        </x:expect>

    </x:scenario>

</x:description>
```

If the iXML grammar is correct then the contents of `1.ixml-output.xml` (the results of the iXML transformation) will now be the same as the contents of the `x:expect` element in Figure 7 [#]. This (temporary) XSpec file can now progress through the normal XSpec steps: compile, evaluate, report and summarise. If the result of the iXML transformation isn't as expected then the differences will be displayed side-by-side in XSpec's HTML report.

## 3.5. What testing revealed

Initially, the errors were caused by a fairly equal mix of syntax errors in my grammar, eg. forgetting to add a full-stop at the end of each rule, and incomplete grammar rules. On the syntax errors front, I resolved this with a more throrough reading of the iXML spec

[2]The name of this function has since changed to `p:invisible-xml`[XPROCIXML]

and comparing other iXML grammars with each other, as well as my own. The incomplete grammar rules were inevitable as I was testing it before it was complete.

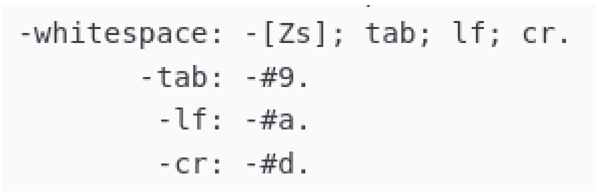
Whitespace was a problem. Michael Sperberg-McQueen wrote about this in 2023, in his paper on *Keyboarding Frege's concept writing*[CMS2023]:

> The handling of whitespace is one of the trickiest and least expected problems confronted by the writer of invisible-XML grammars. Even those with long experience using and writing context-free grammars may be tripped up by it, partly because most practical tools for parser generation assume an upstream lexical analyser or tokenizer which can handle whitespace rules, and most published context-free grammars accordingly omit all mention of whitespace. Because ixml does not assume any upstream lexical analyser, whitespace must be handled by the grammar writer.
>
> If care is not taken, then either whitespace will not be allowed in places where it should be allowed, or it will be allowed by multiple rules, introducing ambiguity into the grammar. (In this case, the ambiguity is usually harmless, since the position of whitespace seldom affects the intended meaning of the input. But there is no way for the parser to know when ambiguity is harmless, so it will warn the user.) On the other hand, if care is taken, then whitespace handling can begin to consume all too much of the grammar writer's thoughts.
>
> —Michael Sperberg-McQueen

That was exactly my experience: finding that whitespace wasn't allowed where it should and then was allowed where it shouldn't. Similarly, before I started writing the grammar, I hadn't really thought about the source including content I might want to drop, for example, delimiters and other punctuation. However, with each new scenario I discovered, I was able to create a new test and if the fix for this new problem unintentionally broke a scenario that was previously working, I knew about it straight away.

Another white-space challenge was understanding what `Zs` is shorthand for. Although it is referenced in the iXML grammar, it's not also defined there, unlike `tab`, `lf` and `cr` which are all defined as well as referenced (see Figure 10 [29].

**Figure 10. iXML definition of whitespace, including Zs**



```
-whitespace: -[Zs]; tab; lf; cr.
       -tab: -#9.
        -lf: -#a.
        -cr: -#d.
```

However, the iXML specification mentions Unicode character classes and, sure enough, `Zs` is defined in *Table 4.4* of the *Character Properties* chapter of the core specification of the Unicode Standard[UNIGEN] (see figure 11). This isn't the only Unicode character class code that's used in the iXML spec so, unless you're familiar with them all, you may also find this table useful.

**Figure 11. Excerpt from the Unicode Standard showing the definition of Zs**

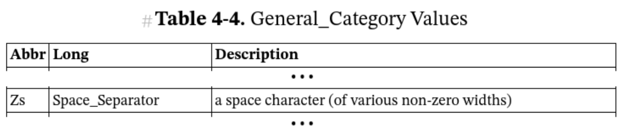
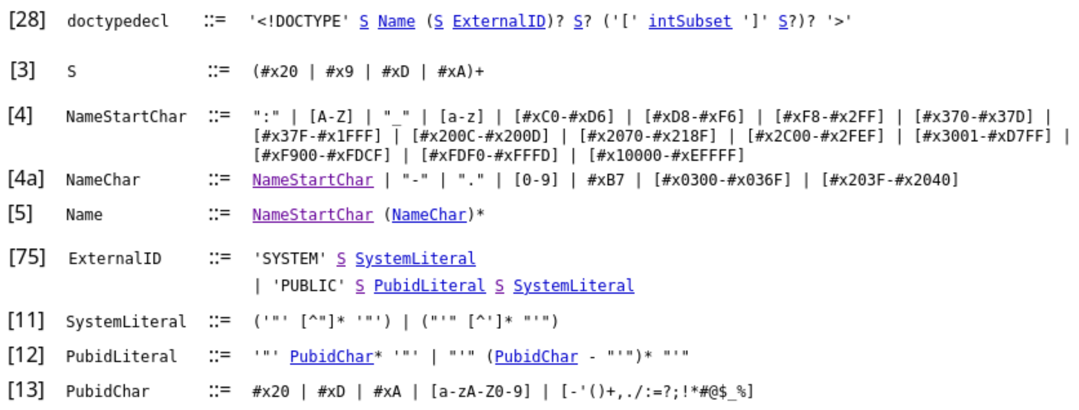


**Table 4-4.** General_Category Values

| Abbr | Long | Description |
|---|---|---|
| | | • • • |
| Zs | Space_Separator | a space character (of various non-zero widths) |
| | | • • • |

As I got towards the end of writing my grammar and I started to try to make sure that it exactly implemented the rules for a DOCTYPE declaration, as defined for XML 1.0 (fifth edition)[XML1-0] and 1.1[XML1-1] (the rules are the same in both specs).

**Figure 12. Excerpts from the XML 1.0 Specification showing the rules for a DOCTYPE declaration**



```
[28]  doctypedecl    ::=  '<!DOCTYPE' S Name (S ExternalID)? S? ('[' intSubset ']' S?)? '>'

[3]   S              ::=  (#x20 | #x9 | #xD | #xA)+

[4]   NameStartChar  ::=  ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6] | [#xF8-#x2FF] | [#x370-#x37D] |
                          [#x37F-#x1FFF] | [#x200C-#x200D] | [#x2070-#x218F] | [#x2C00-#x2FEF] | [#x3001-#xD7FF] |
                          [#xF900-#xFDCF] | [#xFDF0-#xFFFD] | [#x10000-#xEFFFF]
[4a]  NameChar       ::=  NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#x036F] | [#x203F-#x2040]
[5]   Name           ::=  NameStartChar (NameChar)*

[75]  ExternalID     ::=  'SYSTEM' S SystemLiteral
                          | 'PUBLIC' S PubidLiteral S SystemLiteral
[11]  SystemLiteral  ::=  ('"' [^"]* '"') | ("'" [^']* "'")
[12]  PubidLiteral   ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13]  PubidChar      ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]
```

I copied the character ranges from the XML spec and pasted them into my iXML grammar but when I tested the changes, an error was thrown, complaining that character `#EFFFF` wasn't allowed. A quick search revealed that `#EFFFF` is one of 66 Unicode code points confusingly labelled *Noncharacter*[WIKINON][UNINON][UNIGEN]. This prompted me to post a question in the *xml.com* space on Slack.com [https://slack.com]. The conversation this triggered is now lost because messages in that Space are hidden after 90 days (I should have, instead, emailed the question to the iXML mailing list where the discussion would have been preserved) however the kind people of XML Slack helped me understand that:

- the problematic character was in the rule for a Name Start Character
- the iXML definition for a Name Start Character isn't the same as the XML definition for Name Start Character

This isn't the only difference between iXML and XML when it comes to names and the archive for the iXML mailing list includes quite a long discussion about this[CHARDISCUSS]. Norm Tovey-Walsh has also investigated the differences and documented his detailed findings and some history in *Understanding (i)XML names*[IXMLNAMES]. My temporary workaround was to fallback to using the more restrictive iXML definition: `-xml-name-start-character: ["_"; L].`

Another whitespace-related challenge manifested when I configured CoffeePot to pretty print the XML output and XSpec started failing my tests because of whitespace differences. I had already updated the expected result so that it was also pretty printed and the actual result looked identical in the side-by-side diff but on closer examination I discovered that CoffeePot was indenting with spaces, whereas I had used tabs to indent the expected result. The samples are much easier to work with if pretty printed, so I didn't want to remove the indentation but as I believe that tabs are superior to spaces (#religious-war), I wasn't about to change the settings in my text editor to replace tabs with spaces and the odds were low that I would remember to use spaces instead of tabs in just those files. Instead, I created an XSLT identity transform that strips whitespace and added it to my XSpec file as a helper.

The final noteworthy test failure was also XSpec related. Both `x:context` and `x:expect` were set to source their content from external files (`@href`), with no `select` attribute. And it looked as though the result matched the expected, but the tests were failing because the result was an `element()` but expected was a `document-node()`. My workaround for this was to set each of the expects to select the root `element()`, instead of the `document-node()`.

## 4. Automation

I used ANT to connect together the XProc pre-processing step and XSpec's standard steps (compile, evaluate, report, summarise), which are already written in ANT. XSpec actually includes an XProc 1.0 pipeline for running tests (as an alternative to ANT) but my pre-processing step was written using XProc 3.1 - and XProc 1.0 steps are incompatible with XProc 3.0+ steps. It wasn't possible to downgrade the pre-processing step to version 1.0 because it doesn't support the `p:ixml` step, which the pre-processing step depends on.

I used XProc instead of ANT for the pre-processing step because it's usually easier to make changes to an XML file using XProc or XSLT. MorganaXProc-III SE supports the XProc 3.1 `p:ixml` step, for carrying out the iXML transformation, whereas the `invisible-xml` function in XSLT 4.0 isn't yet supported by an XSLT processor. However, the pre-processing step does still include two XSLT transformations. It's so easy to use XSLT within XProc that when writing XProc, if I already know how to achieve something using XSLT, I tend to mix the two together rather than checking to see if a more efficient approach is possible simply using out-of-the-box XProc. Although, once something is working, given time and opportunity, I do often review and reflect on my code and explore whether there are other potentially more suitable options available and refactor to use them instead, eg. `p:add-attribute` and `p:namespace-rename`.

## 5. Conclusion

In my opinion, learning to write an Invisible XML grammar is no more difficult that learning any other schema language and is worth the effort if you have a use-case where it would be useful. Although I would recommend that for your first grammar you too seek out a textual input that already has a very clearly defined, simple, stable and small content model.

It was a relief to confirm that there are viable options other than eyeballing for testing an iXML grammar during development, not least because the notation for an iXML grammar is quite symbol-heavy which can make typos tricky to spot. Tests are also important in the maintenance of a grammar though, so that developers can fix bugs or extend the content model with greater confidence that their changes haven't introduced unintended side-effects. Especially if the person making the change wasn't involved in writing the grammar originally - or even if they were but it's been a long time since they worked on it.

Once the invisible-xml() XPath/XQuery function is more widely supported it will be trivially easy to use XSpec to test an iXML grammar, per the approach proposed by Amanda Galtman. And if you already have the option to do so using XQuery, BaseX and Markup Blitz then you should try it out; it's not complicated.

While my own project has served it's purpose of helping me to learn to write and work with iXML grammars - and better understand how XSpec works under-the-hood - I believe it will be redundant as soon as the invisible-xml() XPath/XQuery function is more widely supported.

## Bibliography

[IXML] Invisible XML Specification 1.0, 20 Jun 2022. W3C Invisible Markup Community Group. https://invisiblexml.org/1.0/

[IMCG] W3C Invisible Markup Community Group. Invisible XML. https://invisiblexml.org/

[PEM2013] Pemberton, Steven. Invisible XML. Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). https://doi.org/10.4242/BalisageVol10.Pemberton01

[XML1-0] Extensible Markup Language (XML) 1.0 (Fifth Edition), 26 Nov 2008. W3C. https://www.w3.org/TR/xml/

[CPOT] Tovey-Walsh, Norm. CoffeePot: An Invisible XML processor, version 3.2.9. NineML. https://docs.nineml.org/current/coffeepot/

[CIR2017] Cirulli, Sandro. XSpec v0.5.0. Presented at XML London 2017, London, England, June 10 - 11, 2017. In XML London 2017 Conference Proceedings. https://xmllondon.com/2017/xmllondon-2017-proceedings.pdf

[GAL2024] Galtman, Amanda. Stretching XPath: Three Testing Tales: Beyond Primary Use Cases of Certain XML Functions and Standards. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Galtman01.

[TDD] Wikipedia contributors. Test-driven development. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Test-driven_development

[WHATXSP] XSpec wiki contributors. What is XSpec. XSpec Wiki. https://github.com/xspec/xspec/wiki/What-is-XSpec

[GALMED] Galtman, Amanda. Testing Invisible XML using XSpec: XPath 4.0 and BaseX Pave the Way. Medium, 7 Aug 2024. https://medium.com/@xspectacles/testing-invisible-xml-using-xspec-e2b11b24b486

[GETST] XSpec wiki contributors. Getting Started. XSpec Wiki.https://github.com/xspec/xspec/wiki/Getting-Started

[XPATH4] XPath and XQuery Functions and Operators 4.0, W3C Editor's Draft, 27 May 2025. W3C. https://qt4cg.org/specifications/xpath-functions-40/

[WRITEXSPEC] XSpec wiki contributors. Writing Scenarios. XSpec Wiki. https://github.com/xspec/xspec/wiki/Writing-Scenarios

[XPROCIXML] XProc 3.1: Invisible XML, Community Group Report, 16 March 2024. W3C. https://spec.xproc.org/master/head/ixml/

[CMS2023] Sperberg-McQueen, C.M. Keyboarding Frege's concept writing: A case study in the use of invisible XML. Presented at Balisage: The Markup Conference 2023, Washington DC, July 31 - August 4, 2023. In Proceedings of Balisage: The Markup Conference 2023. Balisage Series on Markup Technologies, vol. 28 (2023). https://doi.org/10.4242/BalisageVol28.Sperberg-McQueen01

[UNIGEN] Table 4-4. General_Category Values in Chapter 4: Character Properties in The Unicode Standard Version 16.0 – Core Specification, 10 Sept 2024.

The Unicode Consortium. https://www.unicode.org/versions/Unicode16.0.0/core-spec/chapter-4/#G134153

[XML1-1] Extensible Markup Language (XML) 1.1 (Second Edition), 29 Sep 2006. W3C. https://www.w3.org/TR/xml11/

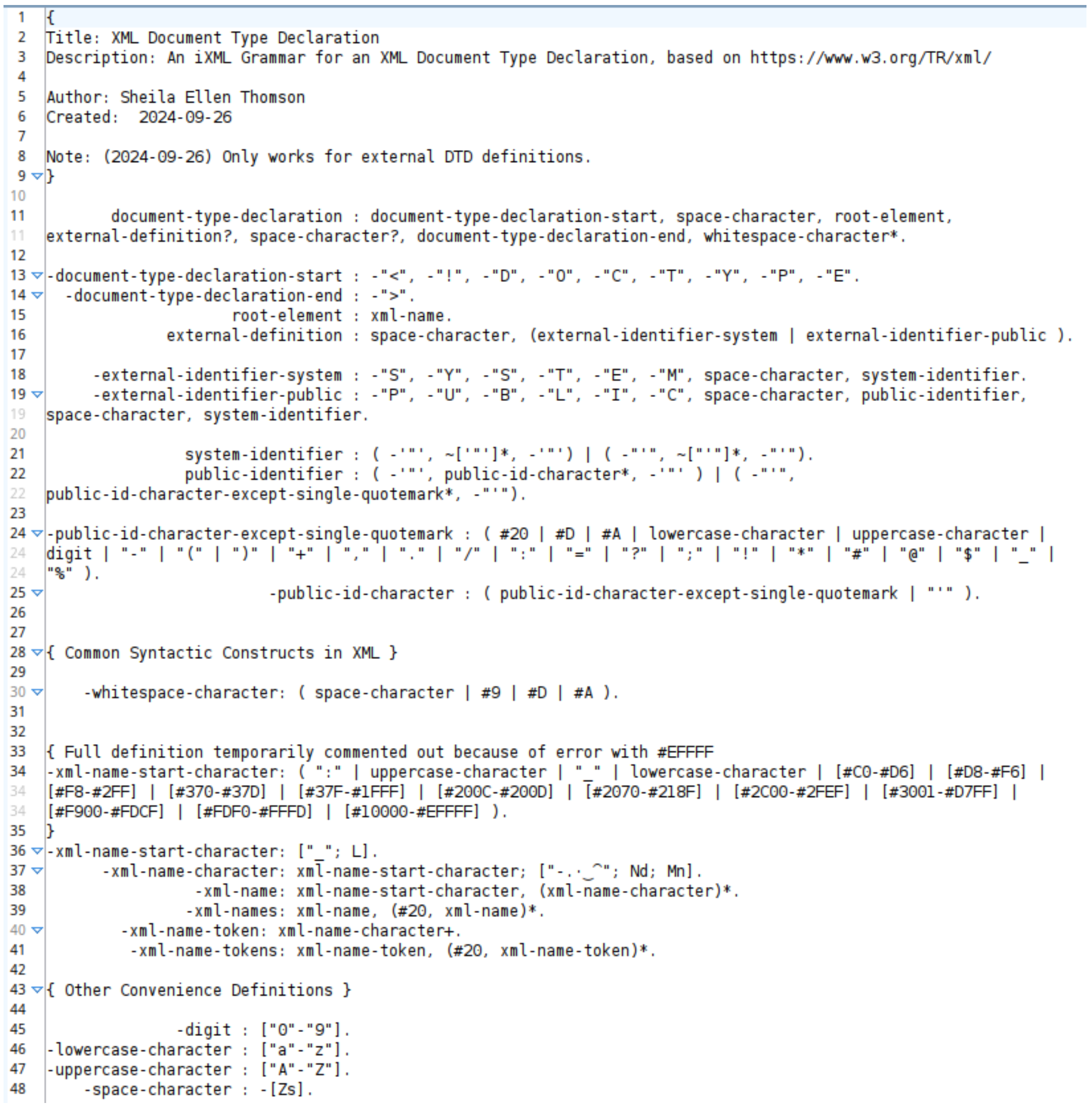
[WIKINON] Wikipedia contributors. Noncharacters section in Universal Character Set Characters. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Universal_Character_Set_characters#Non-characters

[UNINON] Corrigendum #9: Clarification About Noncharacters. The Unicode Consortium. https://www.unicode.org/versions/corrigendum9.html

[CHARDISCUSS] Tovey-Walsh, Norm. An easy-to-miss-error. W3C Invisible XML Community Group Mailing List. 4 Nov 2023. Archive of discussion thread available at: https://lists.w3.org/Archives/Public/public-ixml/2023Nov/0001.html

[IXMLNAMES] Tovey-Walsh, Norm. Understanding (i)XML names. Github.com. https://github.com/invisibleXML/ixml/blob/master/misc/understanding-names.md

## A. Invisible XML Grammar for a DOCTYPE declaration



```
{
Title: XML Document Type Declaration
Description: An iXML Grammar for an XML Document Type Declaration, based on https://www.w3.org/TR/xml/

Author: Sheila Ellen Thomson
Created:  2024-09-26

Note: (2024-09-26) Only works for external DTD definitions.
}

       document-type-declaration : document-type-declaration-start, space-character, root-element, external-definition?, space-character?, document-type-declaration-end, whitespace-character*.

-document-type-declaration-start : -"<", -"!", -"D", -"O", -"C", -"T", -"Y", -"P", -"E".
  -document-type-declaration-end : -">".
                    root-element : xml-name.
             external-definition : space-character, (external-identifier-system | external-identifier-public ).

     -external-identifier-system : -"S", -"Y", -"S", -"T", -"E", -"M", space-character, system-identifier.
     -external-identifier-public : -"P", -"U", -"B", -"L", -"I", -"C", space-character, public-identifier, space-character, system-identifier.

               system-identifier : ( -'"', ~['"']*, -'"') | ( -"'", ~["'"]*, -"'").
               public-identifier : ( -'"', public-id-character*, -'"' ) | ( -"'", public-id-character-except-single-quotemark*, -"'").

-public-id-character-except-single-quotemark : ( #20 | #D | #A | lowercase-character | uppercase-character | digit | "-" | "(" | ")" | "+" | "," | "." | "/" | ":" | "=" | "?" | ";" | "!" | "*" | "#" | "@" | "$" | "_" | "%" ).
                        -public-id-character : ( public-id-character-except-single-quotemark | "'" ).


{ Common Syntactic Constructs in XML }

    -whitespace-character: ( space-character | #9 | #D | #A ).


{ Full definition temporarily commented out because of error with #EFFFF
-xml-name-start-character: ( ":" | uppercase-character | "_" | lowercase-character | [#C0-#D6] | [#D8-#F6] | [#F8-#2FF] | [#370-#37D] | [#37F-#1FFF] | [#200C-#200D] | [#2070-#218F] | [#2C00-#2FEF] | [#3001-#D7FF] | [#F900-#FDCF] | [#FDF0-#FFFD] | [#10000-#EFFFF] ).
}
-xml-name-start-character: ["_"; L].
      -xml-name-character: xml-name-start-character; ["-.·‿⁀"; Nd; Mn].
                -xml-name: xml-name-start-character, (xml-name-character)*.
               -xml-names: xml-name, (#20, xml-name)*.
        -xml-name-token: xml-name-character+.
         -xml-name-tokens: xml-name-token, (#20, xml-name-token)*.

{ Other Convenience Definitions }

              -digit : ["0"-"9"].
-lowercase-character : ["a"-"z"].
-uppercase-character : ["A"-"Z"].
    -space-character : -[Zs].
```

## B. Chronological List of Past Papers on Invisible XML

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

Pemberton, Steven. Data Just Wants to Be Format-Neutral. Presented at XML Prague 2016, Prague, Czech Republic, February 11-13, 2016. In XML Prague 2016 Conference Proceedings. https://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#d6e2656

Pemberton, Steven. Parse Earley, Parse Often: How to Parse Anything to XML. Presented at Presented at XML London 2017, London, England, June 4 - 10, 2016. In XML London 2016 Conference Proceedings. https://xmllondon.com/2016/xmllondon-2016-proceedings.pdf

Pemberton, Steven. On the Descriptions of Data. Presented at XML Prague 2017, Prague, Czech Republic, February 9-11, 2017. In XML Prague 2017 Conference Proceedings. https://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#d6e3022

Pemberton, Steven. On the Specification of Invisible XML Presented at XML Prague 2019, Prague, Czech Republic, February 7-9, 2019. In XML Prague 2019 Conference Proceedings. https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf#page=425

Mason, James David. Do we really want to see markup?. Presented at Balisage: The Markup Conference 2019, Washington, DC, July 30 - August 2, 2019. In Proceedings of Balisage: The Markup Conference 2019. Balisage Series on Markup Technologies, vol. 23 (2019). https://www.balisage.net/Proceedings//vol23/html/Mason01/BalisageVol23-Mason01.html

Sperberg-McQueen, C. M. Aparecium: An XQuery / XSLT library for invisible XML. Presented at Balisage: The Markup Conference 2019, Washington, DC, July 30 - August 2, 2019. In Proceedings of Balisage: The Markup Conference 2019. Balisage Series on Markup Technologies, vol. 23 (2019). https://doi.org/10.4242/BalisageVol23.Sperberg-McQueen01

Hillman, Tom. JayParser: an Invisible XML implementation in XSLT. Presented at Declarative Amsterdam 2020, Amsterdam, Netherlands, October 8-9, 2020. https://declarative.amsterdam/presentations/da.2020.hillman.jayparser

Sperberg-McQueen, C.M. Aparecium, an XQuery / XSLT parser library for invisible XML. Presented at Declarative Amsterdam 2021, Amsterdam, Netherlands, November 4-5, 2021. https://declarative.amsterdam/presentations/da.2021.sperberg-mcqueen.aparecium

Pemberton, Steven. A Pilot Implementation of ixml. Presented at XML Prague 2022, Prague, Czech Republic, June 9-11, 2022. In XML Prague 2022 Conference Proceedings. https://archive.xmlprague.cz/2022/files/xmlprague-2022-proceedings.pdf#page=51

Hillman, Tomos, John Lumley, Steven Pemberton, C. M. Sperberg-McQueen, Bethan Tovey-Walsh and Norm Tovey-Walsh. Invisible XML coming into focus: Status report from the community group. Presented at Balisage: The Markup Conference 2022, Washington, DC, August 1 - 5, 2022. In Proceedings of Balisage: The Markup Conference 2022. Balisage Series on Markup Technologies, vol. 27 (2022). https://www.balisage.net/Proceedings//vol27/html/Eccl01/BalisageVol27-Eccl01.html

Hillman, Tomos, C. M. Sperberg-McQueen, Bethan Tovey-Walsh and Norm Tovey-Walsh. Designing for change: Pragmas in Invisible XML as an extensibility mechanism. Presented at Balisage: The Markup Conference 2022, Washington, DC, August 1 - 5, 2022. In Proceedings of Balisage: The Markup Conference 2022. Balisage Series on Markup Technologies, vol. 27 (2022). https://www.balisage.net/Proceedings//vol27/html/Sperberg-McQueen01/BalisageVol27-Sperberg-McQueen01.html

Sperberg-McQueen, C. M. Keyboarding Frege's concept writing: A case study in the use of invisible XML. Presented at Balisage: The Markup Conference 2023, Washington, DC, July 31 - August 4, 2023. In Proceedings of Balisage: The Markup Conference 2023. Balisage Series on Markup Technologies, vol. 28 (2023). https://doi.org/10.4242/BalisageVol28.Sperberg-McQueen01

Tovey-Walsh, Norm. Ambiguity in iXML: And How to Control It. Presented at Balisage: The Markup Conference 2023, Washington, DC, July 31 - August 4, 2023. In Proceedings of Balisage: The Markup Conference 2023. Balisage Series on Markup Technologies, vol. 28 (2023). https://doi.org/10.4242/BalisageVol28.Tovey-Walsh01

Conrad, Kurt. Word processing is so last century: Formalizing internal narratives using internal declarations and making them look pretty. Presented at Markup UK 2023, London, England, June 1-3. In Markup UK 2023 Proceedings. https://markupuk.org/2023/pdf/Markup-UK-2023-proceedings.pdf#d5e2236

Pemberton, Steven. A Declarative Code Browser with iXML and XForms. Presented at Declarative Amsterdam, Amsterdam, Netherlands, November 2-3, 2023. https://declarative.amsterdam/presentations/da.2023.pemberton.declarative-code-browser

Nordström, Ari. It's Useful After All – VIN Numbers, DITA, and iXML. Presented at XML Prague 2024, Prague, Czech Republic, June 6-8, 2024. In XML Prague 2024 Conference Proceedings. https://archive.xmlprague.cz/2024/files/xmlprague-2024-proceedings.pdf#page=305

Courtney, Joseph Michael, and Michael Robert Gryk. Pulse, Parse, and Ponder: Using Invisible XML to Dissect a Scientific Domain Specific Language. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://www.balisage.net/Proceedings//vol29/html/Courtney01/BalisageVol29-Courtney01.html

Holstege, Mary. Invisible Fish: API Experimentation with InvisibleXML. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Holstege01

Lumley, John. Variations on an Invisible Theme: Using iXML to produce XML to produce iXML to produce …. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Lumley01

Nordström, Ari. Adventures in Mainframes, Text-based Messaging, and iXML. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Nordstrom01

Sperberg-McQueen, C. M. From Word to XML via iXML: a Word-first XML workflow in the TLRR 2e project. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference

2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Sperberg-McQueen01

Tovey-Walsh, Bethan. When women do algorithms: a semi-generative approach to overlay crochet with iXML and XSLT. Presented at Balisage: The Markup Conference 2024, Washington, DC, July 29 - August 2, 2024. In Proceedings of Balisage: The Markup Conference 2024. Balisage Series on Markup Technologies, vol. 29 (2024). https://doi.org/10.4242/BalisageVol29.Tovey-Walsh01

Pemberton, Steven. Banking with iXML and XForms. Presented at Declarative Amsterdam 2024, Amsterdam, Netherlands, November 7-8, 2024. https://declarative.amsterdam/presentations/da.2024.pemberton.banking

Verwer, Nico and Lamers, Pieter. Syntax highlighting for code blocks using iXML. Presented at Declarative Amsterdam 2024, Amsterdam, Netherlands, November 7-8, 2024. https://declarative.amsterdam/presentations/da.2024.verwer.syntax-highlighting-using-ixml

# XForms Extended

## for XSLTForms

Charafeddine Cheraa, Muthabr

The web scene has exploded in the last couple of decades, transitioning from static web pages to dynamic, responsive web applications. Facilitating that are frameworks like React, Vue, Svelte, and Angular that prioritize user experience, component-based architecture, and seamless integration with modern APIs, without sacrificing the developer experience.

Meanwhile, XForms, while powerful for data-centric applications, has for the most part fallen behind. Its development has not caught up with these rapid changes, leaving a growing gap in terms of modern UI/UX capabilities, component flexibility, and integration with the wider web development ecosystem. This disparity is not just a matter of preference; it risks XForms becoming irrelevant.

While the development of XForms 2.0 is ongoing, the progress has been undeniably slow, inviting a different approach to address these issues rather than waiting for the eventual arrival of XForms 2.0. The web is not waiting for XForms, and the momentum behind competing technologies continues to build.

We think that leveraging existing web technologies to address the most pressing limitations of current XForms might be a more beneficial approach.

The idea we came up with is to use a syntax similar to XForms (let's call it Extended XForms for now), which then will be converted to XForms code. This syntax will extend the original one in three ways:

1. **Style extension** this makes minor or no changes to the XML code, but introduces new CSS styles, mostly for layouts, but also for theming when needed.
2. **Basic extension**: the simple and concise syntax of Extended XForms is converted to regular XForms, which then can be used as any other XForms document.

   These extra functionalities can be done in pure XForms but will be complex and difficult to implement and maintain.

   This may also introduce extra JavaScript code for extra functionalities.
3. **Functional extension** this provides additional functionality not currently in XForms, commonly to read data directly from XForms models, write data, and listen to changes.

   Functional extensions depend on the XForms renderer and will not work in the same way for all XForms implementations.

## 1. Introduction

The web scene has exploded in the last couple of decades, transitioning from static web pages to dynamic, responsive web applications. And facilitating that is frameworks like React, Vue, Svelte, and Angular which prioritize user experience, component-based architecture, and seamless integration with modern APIs, without sacrificing the developer experience.

Meanwhile, XForms, while powerful for data-centric applications, has for the most part remained fallen behind. Its development is not catching up with these rapid changes, leaving a growing gap in terms of modern UI/UX capabilities, component flexibility, and integration with the wider web development ecosystem. This disparity is not just a matter of preference; it risks XForms becoming irrelevant.

While the development of XForms 2.0 is still ongoing, the progress has been undeniably slow, which makes taking a different approach to fix this issue sound better than anticipating the eventual arrival of XForms 2.0. The web is not waiting for XForms, and the momentum behind competing technolgies continues to build.

We think that leveraging existing web technologies to address the most pressing limitations of current XForms, might be a more beneficial approach.

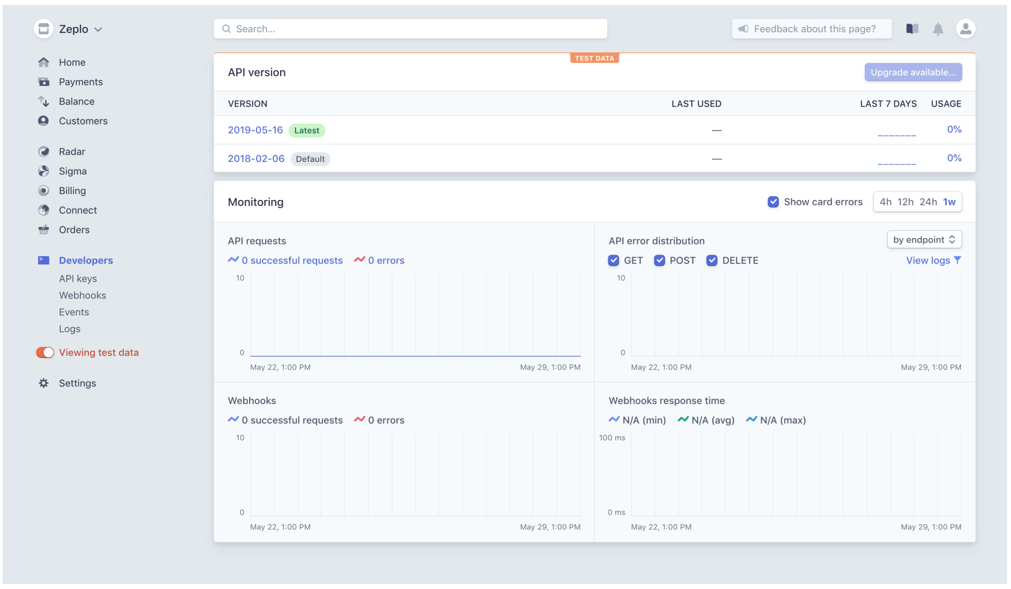
## 2. The problem

XForms is a W3C standard for creating powerful, XML-based web forms that separate data, logic, and presentation. It allows developers to build dynamic, interactive forms with built-in features like data validation, calculations, and conditional logic, all without scripting. XForms is best used in data-intensive applications such as enterprise systems, government portals, and document processing workflows, especially where structured data and XML integration are important.

However, if you try to use it for other applications, issues will start to pop up. Usually, it's either something missing so you can't simply implement it, or something that can be done, but it clearly could be much easier.

Consider the following screenshot as an example to better understand the issues:

Figure 1. Concept design of a SaaS amdin dashboard



| VERSION | LAST USED | LAST 7 DAYS | USAGE |
|---|---|---|---|
| 2019-05-16 Latest | — | | 0% |
| 2018-02-06 Default | — | | 0% |

Can we make this in XForms? Yes.

Will it be in XForms alone? Most likely not.

Will it be easy? Definitely not.

Will the code be straightforward and easy to maintain? No, and it will definitely get more complicated when you need two of the same complex element.

Why is that? And can we avoid it?

## 2.1. Issue #1: Cumbersome layouts

Let's first talk about the layout. We have a side navigation bar, a top bar, and the page content. Both of the sidebar and the top bar are common UI elements that are widely used, as well as the general layout.

This layout can be easily achieved with css:

```
<head>
  <style>
    #app {
      display: flex;
      gap: 4px;
      flex-direction: row;
      height: 100vh;
    }
    #body {
      flex: 1;
      display: flex;
      flex-direction: column;
      gap: 4px;
    }
    #sidebar {
      width: 200px;
    }
    #topbar {
      height: 60px;
    }
    #content {
      flex: 1;
    }
    #sidebar, #topbar, #content {
      font-size: 32px;
      border: 1px solid black;
      padding: 10px;
      line-height: 60px;
    }
  </style>
</head>
<body>
  <div id="app">
    <div id="sidebar">
      <div>Sidebar</div>
    </div>
    <div id="body">
      <div id="topbar">
```

```
        <div>Topbar</div>
      </div>
      <div id="content">
        <div>Content</div>
      </div>
    </div>
  </div>
</body>
```

Then, depending on our use case we may need to add some XForms code to control some aspects of the layout, i.e. the sidebar's width:

```
<div id="sidebar" style="width:{if (instance('app')/sidebar-collapsed ↩
= 'true', 40, 200)}px">
```

This will work, but wouldn't be better if we could write something like:

```
<body>
  <app-layout>
    <sidebar collapsed="instance('app')/sidebar-collapsed = 'true'">
      <div>Sidebar</div>
    </sidebar>
    <topbar height="60">
      <div>Topbar</div>
    </topbar>
    <main>
      <div>Content</div>
    </main>
  </app-layout>
</body>
```

This does not introduce anything that we don't already have; its only appeal is that it is much simpler than the original XHTML/XForms code.

## 2.2. Issue #2: Complex logic

Let's now look at the content of the page. We can see some charts, which also can be made with XForms. For example:

```
<div class="chart">
  <svg width="430" height="290" xmlns="http://www.w3.org/2000/svg">
    <rect width="100%" height="100%" fill="#f8f9fa" />
    <line x1="10" y1="10" x2="10" y2="260" stroke="#333" />
    <line x1="10" y1="260" x2="420" y2="260" stroke="#333" />
    <g fill="#4a90e2">
      <xf:repeat ref="instance('chart')/bars/entry" id="bar">
        <rect x="{-20 + position() * 40}" y="{10 + (100 - .) * 2.5}"  ↩
width="30" height="{. * 2.5}" />
      </xf:repeat>
    </g>
    <g font-size="10" fill="#000" text-anchor="middle">
      <xf:repeat ref="instance('chart')/bars/entry" id="bar">
        <text x="{-5 + position() * 40}" y="275">
          <xf:output value="."/>
        </text>
      </xf:repeat>
    </g>
  </svg>
</div>
```

Again, this code works as intended, but what if we could use this instead:

```
<chart type="bar" data="instance('chart')/bars/entry" />
```

Once again, the appeal of this syntax is its simplicity. What makes it even more appealing is that the chart appears multiple times on this page alone, and this concise syntax will not create a mess when repeated.

### 2.3. Issue #3: The need for advanced JavaScript code

Now imagine that we need to make the previous charts show live data. We simply cannot do it. We can set a JavaScript interval to poll new data from the server, every, let's say, 5 seconds, but then, how are we going to let XForms know what the newly pooled data is?

We can instead use a submission to refresh the data, but how are we going to trigger it from a JavaScript interval?

Actually, we can do both, but it depends on the renderer. For XSLTForms, for example, we can locate the `xforms-model` element dispatch an event from JavaScript, listen for it on XForms, and refresh whenever it's triggered. This works with XSLTForms 1.7 but not with previous versions because they don't build the HTML page in the same way.

```
// javascript code
setInterval(function () {
  document.querySelector('xforms-model').dispatchEvent(new ↩
CustomEvent('on-refresh'));
}, 5000);
```

```
<!-- xforms code -->
<xf:submission id="refresh" method="get" action="/refresh" ↩
replace="instance" instance="test-data"></xf:submission>
<xf:send submission="refresh" ev:event="on-refresh" />
```

However, what if we want a better solution? What if we want to use sockets instead? How complex would implementing that be? Is it worth the trouble, or would it just be a half-baked solution that is slightly easier to implement?

Wouldn't it be much better if we could write something like the following instead:

```
<chart type="bar" data-url="/load-data/bar-chart" refresh- ↩
interval="5" />
```

or

```
<chart type="bar" data-socket="bars-chart" />
```

## 3. The proposed solution:

We came up with Extended XForms to avoid the above issues and try to end up with the best possible XForms user and developer experience.

The idea of Extended XForms is simple: an XSLT stylesheet that transforms the easy-to-understand Extended XForms code to regular XForms so the developer does not need to do everything themselves.

### 3.1. XForms code

*This is still experimental and will most likely change later*

We have a main stylesheet that crawls the whole source of Extended XForms documents.

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

The main stylesheet does not alter the actual code except for `<xhtml:head>`. We output its existing content, and then we add a link to our CSS stylesheet and our JavaScript file.

We then find the main model and append a new instance to it that will hold data managed by Extended XForms.

*In the future, we will probably add a separate model specifically for Extended XForms*

We also apply templates with specific modes, more on this later.

```
<!-- html -->
<xsl:template match="xh:head">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xh:script src="/res/xx-forms.js" type="text/javascript" />
    <xh:link href="/res/xx-forms.css" type="text/css"                ↩
rel="StyleSheet"/>
    <xsl:apply-templates select="node()"/>
    <xsl:apply-templates select="/" mode="head" />
    <xsl:apply-templates select="/" mode="style" />
  </xsl:copy>
</xsl:template>
<!-- model -->
<xsl:template match="xf:model[1]">
  <xsl:copy>
    <xsl:apply-templates select="@*"/>
    <xsl:apply-templates select="node()"/>
    <xf:instance id="xxforms">
      <data>
        <xsl:apply-templates select="/" mode="model" />
      </data>
    </xf:instance>
    <xsl:apply-templates select="/" mode="events" />
  </xsl:copy>
</xsl:template>
```

This alone provides custom CSS classes that can be used directly by the developer. However, most of these classes are meant to be used by Extended XForms. The rest of CSS classes are mostly for theming.

It also provides low-level JavaScript functions allowing:

- Setting XForms data, nodes and attributes.
- Reading XForms data, also nodes or attributes.
- Listening to data changes.

The main stylesheet includes other stylesheets, each for a specific task (a feature, a component, a style, etc.). These stylesheets use the special modes to add their own styles, data, and events.

If needed, the stylesheet can also add a template that matches a specific Extended XForms syntax and replace it with XForms code.

```
<xsl:template match="xxf:sidebar">
...
</xsl:template>
```

### 3.2. JavaScript and CSS code

*This is still experimental and will most likely change later*

Extended XForms has its own JavaScript code that provides the following functionalities:

- Retrieve a model
- Retrieve an instance
- Retrieve a specific data node in an instance
- Retrieve an attribute
- Update a data node
- Track a specific data node's changes.

Unfortunately, these rely heavily on the specific XSLTForms implementation and will not work with XSLTForms versions that use a different implementation, let alone other XForms frameworks.

Even though these functions can be used directly by the developer, they were implemented to be used by the JavaScript code from other Extended XForms stylesheets.

### 3.3. Extended XForms Components

The real extension of XForms happens here. The main stylesheet includes different stylesheets for different extended component, such as tabs, menus, charts, etc.

These components mainly replace their Extended XForms code with proper XForms code. For example, for a video component, the main stylesheet we would have:

```
<xsl:include href="xx-video.xsl"/>
```

And in `xx-video.xsl`:

```
<xsl:template match="xx:video">
  <xhtml:video controls="controls">
    <xsl:attribute name="src">{<xsl:value-of select="@src" />}</        ↩
xsl:attribute>
  </xhtml:video>
</xsl:template>
```

This converts this Extended XForms code:

```
<xx:video src="instance('test-data')/video" />
```

To XForms code:

```
<xhtml:video controls="controls" src="{instance('test-data')/video}"  ↩
/>
```

The component stylesheet can also hook to the template modes applied by the main stylesheet as needed. For example, we can use the `model` mode to add data specific to video components:

```
<xsl:template match="/" mode="model" priority="10">
  <video>
    <curtently-playing>none</curtently-playing>
  </video>
  <xsl:next-match />
</xsl:template>
```

The same can be done with the other modes:

- `model` to add component-specific data.
- `head` to add component-specific script.
- `style` to add component-specific CSS.
- `events` to listen to events.

## 4. Weaknesses and Limitations

This whole idea of extending XForms, as good as it sounds, still has many flaws:

- Much of the project relies heavily on a specific implementation of XForms, XSLTForms 1.7, which is why it will not work with other frameworks (like Orbeon).

  The same principle may or may not be applicable to other implementations. Even if it is, it will need a complete rewrite of the JavaScript code.

- The idea of Extended XForms is to fill the gap between current XForms and the rest of the web, meaning that getting a new version of XForms will heavily affect the project. Most of the code will need to be rewritten, some completely scrapped, and when XForms 2.0 gets to the point that this project is no longer needed, it can be put away for good.

- Although this project helps developers write less code, the final XForms code is still large and complex, and unfortunately we can do little to improve its performance when it starts to decline.

- Extended XForms syntax is very opinionated; it may look good to some, but for others, it may seem like an extra step rather than a shortcut.
- The current implementation uses many CSS and JavaScript files, and although small, they add a bit of a burden on the page's first load. Not to mention that the implementation includes even code that might not be needed in the form being rendered.

## 5. What's next

Extended XForms can be improved in a number of ways:

- Somehow rewrite the layer connecting JavaScript and XForms entirely in XForms code. If doable, Extended XForms will no longer depend on a specific implementation, opening it up to be used by a larger group of developers.
- Improve the transformation to account for what needs to be included (CSS and JavaScript code) to reduce the the extra footprint. Also bundle the code in one file, rather than loading multiple files.
- Streamline the way we write components, allowing for easy updates, easily removing unneeded components, and adding new components. Also, allow the users to write their own components to use in their projects or share with others. This will bring faster updates and help the project not fall behind the rapid growth we are trying to match.
- As for the performance issue, there's only so much we can do. The only way to work around that is to write a new XForms preprocessor/renderer from scratch, which is not a somple task by any means.… but doable... maybe?

## Bibliography

W3C: XForms 1.1 W3C Recommendation. 20 October 2009. https://www.w3.org/TR/xforms11/

XForms Users Community Group: XForms 2.0. https://www.w3.org/community/xformsusers/wiki/XForms_2.0

Alain Couthures: XSLTForms. GitHub repository. https://github.com/AlainCouthures/xsltforms

Alain Couthures: XSLTForms. SourceForge repository. https://sourceforge.net/projects/xsltforms/

Orbeon Forms. https://www.orbeon.com/

Webframe: Concept designs. used for demonstration. https://webframe.xyz/

# Processing JSON with Template Rules

Michael Kay, Saxonica Ltd

This paper describes a case study exploring how effective the current design of XSLT 4.0 is in processing JSON files with a recursive structure, using the rule-based recursive descent design pattern familiar to XSLT users. It explores the conversion of an existing non-trivial XSLT-based application (a transpiler that converts Java source code to C#) to see how well it would be able to cope if the input were JSON rather than XML. The exercise has led to a number of changes in the proposed features of XSLT 4.0, and identifies other areas where the design could be further improved.

## 1. Introduction

XSLT 3.0, together with its accompanying specifications such as XPath 3.1, introduced support for processing and generating JSON alongside XML. The new features have proved useful, but they have known limitations.

Saxonica, for example, uses the JSON capabilities in XSLT 3.0 when processing new online orders from customers for the Saxon product. We use a third-party service, Ecwid, that supplies details of new orders in JSON format, and we use an XSLT application to process this order, add the details to our XML orders database, and generate license keys and email notifications to the customer. The application uses XForms and SaxonJS. It pulls the JSON information from Ecwid using a call on `fn:json-doc` with an appropriate URI, and then extracts the required data using path expressions such as

```
<xsl:variable
      name="subscriptionOption"
      select="$items?items?1?recurringChargeSettings"
      as="map(*)?" />
```

The JSON structure is straightforward, and the features in XSLT 3.0 and XPath 3.1 are more than adequate to handle it. [1]

But the capabilities of XSLT for processing JSON are more limited than the capabilities for processing XML. One of these limitations, the one addressed in this paper, is the ability to transform JSON using XSLT's quintessential processing model: rule-based recursive-descent transformation using template rules.

A project is currently underway, informally known as QT4, to define 4.0 versions of the XSLT, XPath, and XQuery languages. This project has been set up as a W3C Community Group and meets weekly to discuss and agree proposed changes to the specification. The activity can be tracked online at `https://qt4cg.org/` and of course anyone is welcome to participate. To date over 500 changes to the specifications have been accepted, and

[1]More details of this application can be found at [Delpratt and Lockett 2017]. At the time of that paper the Ecwid data feed was plain text rather than JSON, but the paper does describe some other ways in which the application uses JSON internally.

most of these have been implemented in Saxon and/or BaseX; more than 12,000 test cases have been added to the XQuery test suite alone, on top of the 32,000 test cases already available for the 3.1 specifications[2].

But I was concerned that we hadn't really tackled or solved the issues concerned with recursive-descent transformation. Back in 2016, before XSLT 3.0 was even finalised, I published a paper [Kay 2016] at XML Prague in 2016 giving a couple of worked examples of JSON transformations using XSLT 3.0, coming to the rather unhappy conclusion that they were best tackled by converting the JSON to XML, transforming the XML, and then converting the XML back to JSON. I returned to these examples in a Balisage paper in 2022 [Kay 2022] where I showed that these two particular problems could be tackled much more easily using new features proposed for XSLT 4.0; however I remained uneasy that neither of the two problems really featured the recursive-descent processing paradigm.

So I resolved to conduct a case-study in which I would select a realistic application in which recursive-descent rule-based transformation of JSON input was a requirement, and use this application to test the usability of the XSLT 4.0 specifications in their current state, and propose enhancements where they were found to be necessary. This paper summarises the conclusions of that study. A blow-by-blow account containing contemporaneous notes of the tasks undertaken can be found at `https://github.com/qt4cg/qtspecs/issues/1786`; this paper focuses more on the final conclusions, and ignores some of the avenues I followed that produced no useful insights.

## 2. Selecting the case study

The two key criteria for selecting a case study were (a) that use of JSON (rather than XML) should be a natural choice for the input and output data, and (b) that the data should be recursive — because it's with recursive data structures that the recursive-descent processing model becomes a necessary part of the solution.

The application that best fitted these requirements was the Java-to-C# transpiler, which I described at Markup UK in 2021: [Kay 2021] This is an application that is written almost entirely in XSLT (with a small amount of control logic in Java and Gradle). It is a live application, used internally by Saxonica on a daily basis to transform our Java source code into C# source code, which is then used to build the SaxonCS product. The application runs in several phases:

1. We preprocess the Java code using a Java preprocessor to exclude parts of the code that are are not needed in SaxonCS.

   We run the open-source `JavaParser` product to generate an XML representation of the syntax tree of each (preprocessed) Java module in the Saxon product. This produces around 110Mb of XML across 2100 files.

2. We analyse these files to produce a digest file. The digest contains a list of classes, interfaces, and methods across the product as a whole, in a single XML file. The digest is around 4Mb.

3. We refine the digest file, producing a modified version with augmented information. The main purpose of this process is to work out which Java methods are overridden, so that in the generated C#, the can be suitably annotated with `virtual` or `override` modifiers, something that is not possible by looking at each Java module in isolation.

4. We then transform each of the XML modules into a C# serialization, taking account of information in the digest file. This stage is a pure recursive-descent rule-based

[2]Data obtained, naturally, using Saxon and XQuery.

transformation, using around 350 template rules to handle each of the syntactic constructs identified by the Java parser.

As written, the application doesn't use JSON. But what if it did? It's convenient that the JavaParser product generates XML, but it didn't have to be that way: JSON would work just as well. There's no mixed content, which is the key feature that would give XML a natural advantage.

So the case study pretends that we're starting with a syntax tree in JSON rather than XML; and furthermore, it explores the use of JSON (rather than XML) for the digest file. Rather than converting the JavaParser to emit JSON, however, we start by converting the XML to JSON, which also gives us a chance to test the new features in XSLT 4.0 for converting XML to JSON.

Some might argue that using XSLT for converting Java to C# is not exactly a typical use case for XSLT. That's a fair criticism. However, I've seen XSLT used for many applications that you might not consider typical: for example, converting the output of a CAD tool into instructions controlling a 3-D printer. Wherever complex data needs to be structurally transformed, XSLT is a possible solution.

Note that it wasn't an aim of the case study to produce a complete working application. Rather, the aim was to identify whether this was likely to be feasible, and what difficulties might be encountered, and how proposed new language features might mitigate any problems.

The remaining sections of the paper focus on what we learned examining each part of the application.

## 3. Converting the input XML to JSON

This was an opportunity to try out the new `fn:element-to-map` function. I described the basic design for this function in a paper at Balisage in 2023[Kay 2023], and we had an implementation in Saxon ready to test.

The function doesn't convert lexical XML to lexical JSON: rather, it converts the XDM representation of XML to the XDM representation of JSON (XDM being the X data model that underpins XSLT and XQuery).

The idea of the function is that it converts each element type in one of a dozen or so different ways, depending on the content model. The content model can be inferred either from a schema, or from examination of a sample collection of input documents, or from an individual element instance. I didn't try to do a schema-aware conversion because (a) no schema was available, and (b) generating one wouldn't be particularly useful, because of the way the particular XML vocabulary works. Specifically, a typical fragment of the XML (representing the Java code `if (iter.next() != null) {iter.close(); return BooleanValue.FALSE}`) looks like this:

```
<statement nodeType="IfStmt">
   <condition nodeType="BinaryExpr" operator="NOT_EQUALS">
      <left nodeType="MethodCallExpr" >
         <name nodeType="SimpleName" identifier="next"/>
         <scope nodeType="NameExpr">
            <name nodeType="SimpleName" identifier="iter"/>
         </scope>
      </left>
      <right nodeType="NullLiteralExpr"/>
```

```
    </condition>
    <thenStmt nodeType="BlockStmt">
       <statements>
          <statement nodeType="ExpressionStmt">
             <expression nodeType="MethodCallExpr" >
                <name nodeType="SimpleName" identifier="close"/>
                <scope nodeType="NameExpr">
                   <name nodeType="SimpleName" identifier="iter"/>
                </scope>
             </expression>
          </statement>
          <statement nodeType="ReturnStmt">
             <expression nodeType="FieldAccessExpr">
                <name nodeType="SimpleName" identifier="FALSE"/>
                <scope nodeType="NameExpr">
                   <name nodeType="SimpleName"
                         identifier="BooleanValue"/>
                </scope>
             </expression>
          </statement>
       </statements>
    </thenStmt>
 </statement>
```

The element names here (`left`, `right`, `condition`, `thenStmt`) tell you nothing about what kind of thing the element is (and therefore what kind of structure its content has); rather it tells you about where it fits into the structure of the parent element. It's the `nodeType` attribute that tells you about the content model: if `nodeType="condition"` then there will be children named `left` and `right`, while if `nodeType="IfStmt"` then there will be children named `condition`, `thenStmt`, and optionally `elseStmt`.

This design, as well as making it difficult to construct a schema, also makes it difficult for the `element-to-map` function to infer the right XML-to-JSON mapping to use for each element name.

Another consequence of the design is that most of the transpiler consists of rules of the form `match="*[@nodeType='MethodCallExpr']"`: it is the `nodeType` attribute that drives the processing, not the element name.

The `elements-to-maps()` function, as it was specified and implemented at the time, had an option `'uniform':true()` that caused the function to analyze the entire input and infer a mapping for each element name that took into account all the elements encountered with that name. By running this against the entire collection of 2100 input files, it ended up making quite reasonable decisions, so far as one could tell. However, for constructs that only appeared very rarely, it might have made a poor choice, and I probably wouldn't have noticed because, in absence of a complete implementation of the transpiler, we didn't get as far as testing that we were generating correct C# code at the end of the process.

It also became clear that examining all the structures that occur in the input to the function doesn't necessarily give the right answer if you run the same conversion on a different set of input files the following day. Because there is downstream code processing the JSON output, it is vital that tomorrow's output is consistent with today's.

This experience led to a decision to make the choices made by the processor more visible and open to scrutiny and adjustment. We split the function into two: `element-to-map-plan()` takes a corpus of XML documents and generates a conversion plan, specifically a so-called "layout" to be used for each element name. The second function, `element-to-map()`, accepts this plan as input, and uses it to guide a specific conversion. The plan is designed so it can itself be serialized as JSON, which means that (a) it can be modified by hand, and (b) the same plan can be used consistently every time a conversion is run, even though the original data is no longer available.

The JSON version of the XML fragment shown above ends up looking like this:

```
{"_nodeType":"IfStmt",
  "condition":{"_nodeType":"BinaryExpr",
    "_operator":"NOT_EQUALS",
    "left":{"_nodeType":"MethodCallExpr",
      "name":{"_nodeType":"SimpleName",
        "_identifier":"next"
      },
      "scope":{"_nodeType":"NameExpr",
        "name":{"_nodeType":"SimpleName",
          "_identifier":"iter"
        }
      }
    },
    "right":{"_nodeType":"NullLiteralExpr"
    }
  },
  "thenStmt":{"_nodeType":"BlockStmt",
    "statements":[{"_nodeType":"ExpressionStmt",
        "expression":{"_nodeType":"MethodCallExpr",
          "name":{"_nodeType":"SimpleName",
            "_identifier":"close"
          },
          "scope":{"_nodeType":"NameExpr",
            "name":{"_nodeType":"SimpleName",
              "_identifier":"iter"
            }
          }
        }
      },

      {"_nodeType":"ReturnStmt",
        "expression":{"_nodeType":"FieldAccessExpr",
          "name":{"_nodeType":"SimpleName",
            "_identifier":"FALSE"
          },
          "scope":{"_nodeType":"NameExpr",
            "name":{"_nodeType":"SimpleName",
              "_identifier":"BooleanValue"
            }
          }
        }
      }
    ]
  }
}
```

I decided to use "_" rather than "@" as a prefix for JSON properties derived from XML attributes, on the grounds that the result is a valid NCName and can therefore be more easily selected using the XPath lookup operator, for example `$node?_nodeType`. The `element-to-map` function allows any prefix (or none) to be used.

The fragments shown above illustrate the XML and JSON representations of the raw Java parse tree. In practice the JavaParser product also has an option (the type solver) to decorate the parse tree with additional attributes containing the inferred types of various constructs, and their expanded names. For example the `left` node of the first `condition` above has two additional properties: `"_RETURN": "net.sf.saxon.value.AtomicValue"`, and `"_RESOLVED_TYPE": "com.saxonica.functions.qt4.DuplicateValues.DuplicatesIterator"`, indicating that in the Java method call `iter.next()`, the type of `iter` is `com.saxonica.functions.qt4.DuplicateValues.DuplicatesIterator`, and the return type of the method call is `net.sf.saxon.value.AtomicValue`.

## 4. Serializing the parse tree

In the real transpiler, the final stage of processing is to take each of the XML (now JSON) documents representing the parse tree of a module, and, with the aid of information in the digest file, to generate corresponding C# code. This combines two tasks: handling any differences between Java and C#, and then serializing the result (with sufficient indentation and spacing to make it legible, since we're going to need to debug it).

For the sake of the case study, I decided to skip the business logic of Java to C# conversion, and simply re-serialize the parse tree as Java code. This mirrored the development approach I had used for the transpiler, where I first wrote template rules to convert the parse tree back to Java, and then incrementally modified the XSLT to handle cases where the C# needed to be different.

I didn't attempt to rewrite all the template rules, but converted a sufficient subset that several of the larger Java modules could be successfully processed. I felt this would give us all the feedback we needed on whether the task was feasible.

A typical (but very simple) template rule in the transpiler might look like this:

```
<xsl:template match="*[@nodeType='ReturnStmt']">
    <xsl:call-template name="indent"/>
    <xsl:text>return </xsl:text>
    <xsl:apply-templates select="*"/>
    <xsl:text>;{$NL}</xsl:text>
</xsl:template>
```

This rule processes an expression with `@nodeType='ReturnStmt'` and outputs the (Java or C#) text `"return XXX;"` with suitable indentation, and followed by a newline. The XXX here is constructed by recursive application of template rules to the single operand of the return statement (if any): `select="*"` selects the operand, whatever it might be, and processes it using its own template rule.

The rule doesn't need much changing to handle JSON instead of XML. It becomes:

```
<xsl:template match=".[?_nodeType='ReturnStmt']">
    <xsl:call-template name="indent"/>
    <xsl:text>return </xsl:text>
    <xsl:apply-templates select="?expression"/>
    <xsl:text>;{$NL}</xsl:text>
</xsl:template>
```

Some observations:

- `match="."` matches anything. We could have written `match="map(*)"` to indicate that we're only interested in matching maps; or we could have written `match="record(_nodeType, *)` to indicate that we're only interested in matching maps having a `"_nodeType` property. I quite like the idea of combining that with the predicate to allow syntax like `match="record(_nodeType='ReturnStmt', *)"` but that's wishful thinking for now.

- The `xsl:text` instruction produces a text node. The stylesheet as a whole is producing a text file (Java or C# source code), and the traditional way of doing that is to use the XSLT text output method, with a result tree consisting entirely of text nodes. There's a lot of inbuilt XML legacy there, but it works.

  The variable `{$NL}` is used in preference to a literal newline because it doesn't disrupt the indentation of the code. This is purely a matter of personal style.

  Using the latest features in the XSLT 4.0 spec, we could replace the last three lines in the template body with `<xsl:text>return {apply-templates(?expression)};{$NL}</xsl:text>`, which some people might prefer.

- The original XSLT uses `select="*"` in the `apply-templates` instruction to select all children; the revised XSLT uses `select="?expression"` to select only the `expression` child. That is because the attributes and children of the element node in the XML have all become named properties in the JSON, and `?*` would select them all. There's no convenient way with a lookup expression of saying something like `select="?* except ?_nodeType"` (the XPath `except` operator only works with nodes). We have an open issue on this.

It turns out to be rather convenient that we can define the match patterns of template rules based on the properties of a map in the JSON, rather than on the associated key. If instead of `"right":{"_nodeType":"NullLiteralExpr"}` we had to cope with `"NullLiteralExpr":{"_role":"right"}` (a design that could equally well have been chosen), then the matching would become rather more complex, as we shall see.

While most of the template rules in this stylesheet match on the value of the `nodeType` attribute, this isn't true of all of them.

- With the JSON tree, there's no obvious equivalent of `match="/"`, which matches the root of the tree. There's a good reason for this: the XDM model for JSON doesn't include parent pointers, so a map or array that's at the top of the tree produced by parsing JSON doesn't actually *know* that it's at the top of the tree.

  With this example, we know that the JSON is in the form of a singleton map with the key `root`: that is, the JSON starts with:

  ```
  { "root":{
      "_nodeType": "CompilationUnit",
  ```

```
        "packageDeclaration": {
```

and we can take advantage of this by using `match="record(root)"` to match the outermost map.

- In other cases where the original stylesheet matched on element name, it was usually possible to exploit redundancy in the data to match on properties instead. For example the element with name `packageDeclaration` always has the attribute `nodeType="packageDeclaration"`, and the element with name `imports` contains a sequence of elements each having the attribute `nodeType="importDeclaration"`.

  In one or two cases a template rule that matched on an element name (to handle a particular part of an expression, such a the `finally` clause of a try/catch) could simply be inlined into the calling template.

The conclusion from this exercise was that the conversion to handle JSON rather than XML input was straightforward — but that we had been lucky. The template rules all matched on attribute values rather than element names; and none of them made use of features such as XML node identity, or access to parents, ancestors, or siblings, that would be difficult to replicate in the JSON world.

Also: I've glossed over the fact that in this phase, I was merely looking at the code that serializes the parse tree back to Java, and skipped the “business logic” that does the conversion from Java to C#. That code, from a fairly superficial examination, includes a few things that are rather harder to deal with:

- The template rules access information from the digest file using an `xsl:key` definition. The key definition is essentially the same as that described in the subsequent section *Refining the digest file*, and creates the same challenges.
- There are a number of functions and templates that use the parent or attribute axis to examine the context of an expression. For example there is a function `isInterfaceMember` that distinguishes methods defined in a class from methods defined in an interface, which it does by searching the ancestor axis to see whether the containing type is a class or an interface. With a JSON model there are always two ways of tackling this: the needed context information (class or interface?) can be passed down the call tree as a tunnel parameter, or the mechanism for pinning the tree can be used to expose an equivalent to the ancestor axis. This is discussed further in a later section.

## 5. Generating the digest file

Let's look at another stage of the transpilation process: generation of the digest file. In the existing transpiler, this reads the entire collection of 2100 XML files produced by the Java parser, and constructs a single XML file (the digest) containing summary details of the classes, interfaces, and methods. Here is a short extract (the real thing is about 71,000 lines):

```
<digest>
   <module package="net.sf.saxon.tree">
      <class name="NamespaceNode">
         <implements name="net.sf.saxon.om.NodeInfo"/>
         <constructor params="net.sf.saxon.om.NodeInfo|                    ↩
net.sf.saxon.om.NamespaceBinding|int"/>
```

```
         <field name="element" type="net.sf.saxon.om.NodeInfo"/>
         <field name="nsBinding"                                        ↩
type="net.sf.saxon.om.NamespaceBinding"/>
         <field name="position" type="int"/>
         <field name="fingerprint" type="int"/>
         <method name="getTreeInfo"                                     ↩
returns="net.sf.saxon.om.TreeInfo"/>
         <method name="head" returns="net.sf.saxon.om.NodeInfo"         ↩
csReturns="net.sf.saxon.om.Item"/>
         <method name="getNodeKind" returns="int"/>
         <method name="equals" returns="boolean"                        ↩
sig="java.lang.Object" params="java.lang.Object"/>
         <method name="hashCode" returns="int"/>
         <method name="getSystemId" returns="java.lang.String"/>
         <method name="getPublicId" returns="java.lang.String"/>
         <method name="getBaseURI" returns="java.lang.String"/>
         <method name="getLineNumber" returns="int"/>
         <method name="getColumnNumber" returns="int"/>
         ...
```

The JSON equivalent, which we will be generating here, mirrors this closely:

```
 { "digest":[
    {
      "package": "net.sf.saxon.tree",
      "class": [
        { "name":"NamespaceNode" },
        { "implements":{ "name":"net.sf.saxon.om.NodeInfo" } },
        { "constructor":{ "params":"net.sf.saxon.om.NodeInfo|           ↩
net.sf.saxon.om.NamespaceBinding|int" } },
        { "field":{ "name":"element",                                   ↩
"type":"net.sf.saxon.om.NodeInfo" } },
        { "field":{ "name":"nsBinding",                                 ↩
"type":"net.sf.saxon.om.NamespaceBinding" } },
        { "field":{ "name":"position", "type":"int" } },
        { "field":{ "name":"fingerprint", "type":"int" } },
        { "method":{ "name":"getTreeInfo",                              ↩
"returns":"net.sf.saxon.om.TreeInfo" } },
        { "method":{ "name":"head",                                     ↩
"returns":"net.sf.saxon.om.NodeInfo",                                   ↩
"csReturns":"net.sf.saxon.om.Item" } },
        { "method":{ "name":"getNodeKind", "returns":"int" } },
        { "method":{ "name":"equals", "returns":"boolean",              ↩
"sig":"java.lang.Object", "params":"java.lang.Object" } },
        { "method":{ "name":"hashCode", "returns":"int" } },
        { "method":{ "name":"getSystemId",                              ↩
"returns":"java.lang.String" } },
        { "method":{ "name":"getPublicId",                              ↩
"returns":"java.lang.String" } },
        { "method":{ "name":"getBaseURI",                               ↩
"returns":"java.lang.String" } },
        { "method":{ "name":"getLineNumber", "returns":"int" } },
        { "method":{ "name":"getColumnNumber",                          ↩
"returns":"int" } },
```

Actually, what I'm showing here is the result of converting the XML digest to JSON using the 4.0 `element-to-map()` function. In this stage of the case study, we're looking at the code needed to generate this structure, but I didn't actually complete the exercise, partly because the code uses features not yet implemented in Saxon.

The stylesheeet is fairly small (just 180 lines). It uses a mode with `on-no-match="fail"` so there has to be an explicit template rule for every element of interest. The top two templates (in the XML version) are:

```
<xsl:template name="xsl:initial-template">
   <digest>
      <xsl:apply-templates select="collection($xmlDir ||
                                  '?recurse=yes;select=*.xml')"/>
   </digest>
</xsl:template>

<xsl:template match="root">
   <module package="{f:qualifiedName(packageDeclaration/name)}">
      <xsl:apply-templates select="types/type"/>
   </module>
</xsl:template>
```

So the entry-point template reads all the XML files in a directory whose name is supplied as a parameter, and invokes a template to process each file independently; this selects the only elements of interest, which are the type elements (a type being typically a class or interface). These two templates translate in the JSON version to:

```
<xsl:template name="xsl:initial-template">
   <xsl:map>
      <xsl:map-entry key="'digest'">
         <xsl:array>
            <xsl:for-each select="(collection($jsonDir ||
                           '?recurse=yes;select=*.json') ! pin(.)) ?  ↩
*">
               <xsl:array-member>
                  <xsl:map>
                     <xsl:apply-templates select="."/>
                  </xsl:map>
               </xsl:array-member>
            </xsl:for-each>
         </xsl:array>
      </xsl:map-entry>
   </xsl:map>
</xsl:template>

<xsl:template match="?root">  <!-- match=".[label(.)?key = 'root']"  ↩
-->
   <xsl:map-entry key="'module'">
      <xsl:map>
```

```
          <xsl:map-entry key="'_package'"
                         select="f:qualifiedName(?packageDeclaration? ↩
name)"/>
          <xsl:apply-templates select="?types?type"/>
       </xsl:map>
    </xsl:map-entry>
 </xsl:template>
```

Observations:

- The JSON version of the digest is a singleton map, with key `"digest"`, whose value is an array of maps. Constructing this top-level map-of-array-of-maps is somewhat verbose, but straightforward enough.
- The first template rule applies the `pin()` function to each of the 2100 XML documents in the collection, before applying templates to the result. I'll have more to say about the `pin()` function in due course, what it does is to create a copy of the tree of maps and arrays, with each item in the tree augmented with a label carrying information about where it was found in the tree.

  It's possible we may decide that when `xsl:apply-templates` selects a map or array, it should be pinned automatically. That decision hasn't been made yet.
- The second template rule is shown with two alternative forms of the match pattern. The commented-out version works in Saxon today: it tests whether the label of the item (created when it was pinned) has a `key` property of `root`. The second form is a proposed contraction: `match="?root"` is proposed syntax equivalent to the first form. This would only work if the tree has been pinned, because without this, a value in the tree knows nothing about its associated key. (Contrast this with the XDM model for XML, where an element name is an intrinsic property of an element node.)

This stylesheet processes the input JSON by applying templates to the values found in its arrays and maps: the default processing is `<xsl:apply-templates select="?*"/>`. This selects the values, not the key-value pairs. I have experimented with selecting key value pairs instead, using `<xsl:apply-templates select="map:entries(.)"/>`, and there are some cases where this is a good solution, but I have usually found it causes confusion. If the values are labelled with their associated key (by pinning the tree before we start), then it turns out not to be necessary.

For many of the functions and templates in the stylesheet, the translation is fairly direct. For example the XML version has a function to test whether a Java interface has an annotation marking it as a functional interface [3]:

```
<xsl:function name="f:isDelegate" as="xs:boolean">
   <xsl:param name="interfaceDecl" as="element()"/>
   <xsl:choose>
      <xsl:when test="$interfaceDecl/annotations/
                      annotation/name/@identifier='CSharpDelegate'">
         <xsl:sequence select="$interfaceDecl/annotations/
                      annotation[name/@identifier='CSharpDelegate']/
                      memberValue/@value = 'true'"/>
```

[3]In the Java source code, we use the annotation `@CSharpDelegate` to mark interfaces that should be transpiled to C# delegates. The JavaParser faithfully copies this annotation into the XML syntax tree, and the transpiler picks it up from there.

```
       </xsl:when>
       <xsl:otherwise>
          <xsl:sequence select="exists($interfaceDecl
                       [@nodeType='ClassOrInterfaceDeclaration']
                       [@isInterface='true']
                       [annotations/annotation/name/                   ↩
@identifier='FunctionalInterface']
                       [count(members/member)=1])"/>
       </xsl:otherwise>
    </xsl:choose>
 </xsl:function>
```

In the JSON version this becomes:

```
 <xsl:function name="f:isDelegate" as="xs:boolean">
    <xsl:param name="interfaceDecl" as="item()"/>
    <xsl:choose>
       <xsl:when test="$interfaceDecl?annotations
                       ?*?name?_identifier='CSharpDelegate'">
          <xsl:sequence select="$interfaceDecl?annotations
                       ?*[?name?_identifier='CSharpDelegate']
                       ?memberValue?_value = 'true'"/>
       </xsl:when>
       <xsl:otherwise>
          <xsl:sequence select="exists($interfaceDecl
                       [?_nodeType='ClassOrInterfaceDeclaration']
                       [?_isInterface='true']
                       [?annotations?*?name?                           ↩
_identifier='FunctionalInterface']
                       [count(?members?*)=1])"/>
       </xsl:otherwise>
    </xsl:choose>
 </xsl:function>
```

which, although it might appear a little cryptic at first sight, is actually a very direct translation.

Incidentally, both versions could take advantage of the new `xsl:if` instruction in XSLT 4.0: the JSON version could be written:

```
<xsl:function name="f:isDelegate" as="xs:boolean">
    <xsl:param name="interfaceDecl" as="item()"/>
    <xsl:if test="$interfaceDecl?annotations
                  ?*?name?_identifier='CSharpDelegate'"
            then="$interfaceDecl?annotations
                  ?*[?name?_identifier='CSharpDelegate']
                  ?memberValue?_value = 'true'"
            else="exists($interfaceDecl
                  [?_nodeType='ClassOrInterfaceDeclaration']
                  [?_isInterface='true']
```

```
                 [?annotations?*?name?                            ↩
_identifier='FunctionalInterface']
                 [count(?members?*)=1])"/>
 </xsl:function>
```

This stylesheet outputs JSON, and it is therefore greatly concerned with constructing maps and arrays. We've already seen in the top-level template that this can be rather verbose. There's a lot of this kind of code:

```
 <xsl:map>
    <xsl:map-entry key="'name'" select="f:degenerify(?name?        ↩
_identifier)"/>
    <xsl:if test="f:isDelegate(.)">
       <xsl:map-entry key="'delegate'" select="1"/>
    </xsl:if>
    ...
    <xsl:map-entry key="'members'">
       <xsl:array>
          <xsl:for-each select="?members?*[?                      ↩
_nodeType='MethodDeclaration']">
             <xsl:array-member>
                <xsl:apply-templates select="."/>
             </xsl:array-member>
          </xsl:for-each>
       </xsl:array>
    </xsl:map-entry>
 </xsl:map>
```

and it would be nice to reduce the verbosity if we can. One way of doing this is to do more of the work in XPath expressions rather than XSLT instructions. A couple of new XSLT features are designed to facilitate this: an `xsl:select` instruction, which evaluates an XPath expression held in its content, and an `apply-templates` function, which does the same thing as the `xsl:apply-templates` instruction. With these enhancements, we can almost replace the above code by:

```
  <xsl:select> {
    'name' : f:degenerify(?name?_identifier),
    'delegate' : xs:integer(f:isDelegate(.)),
    ...
    'members' : array:build(?members?*[?                          ↩
_nodeType='MethodDeclaration'],
                            apply-templates#1)
  } </xsl:select>
```

The only ingredient missing here is that map constructors have no way to generate a map entry (such as `'delegate'`) conditionally. We're working on that one.

Another attempt to make map construction more concise is a new `xsl:record` instruction, allowing something like:

```
<xsl:record
  name = "f:degenerify(?name?_identifier)"
  delegate = "xs:integer(f:isDelegate(.))"
  ...
  members = "array:build(?members?*[?_nodeType='MethodDeclaration'],
                        apply-templates#1)"/>
```

Again, it offers no way to generate a map entry conditionally.

## 6. Refining the digest

The final part of the transpiler to be examined in this case study is the stylesheet that refines the digest. This is concerned with adding attributes to the information about classes and methods: the most obvious example is to annotate C# methods as `virtual` if they are overridden in a subclass, or as `override` if they are overriding a method in a superclass. Another task is to change the return type of a method if it overrides a superclass method with a wider return type: when we wrote the transpiler, C# did not allow covariant return types, and it still imposes restrictions that are more severe than those in Java (for example for methods defined in interfaces).

By its nature, this stylesheet is often following links from the usage of a class to the definition of the class, and this is achieved using an XSLT key definition:

```
 <xsl:key name="classKey"
          match="class | interface"
          use="ancestor::module/@package || '.' ||
                 string-join(ancestor-or-self::*
                         [self::class|self::interface]/@name,       ↩
'.')"/>
```

This indexes every class or interface by a key that represents the full hierarchic name of the class or interface. For example, given this structure:

```
 <module package="net.sf.saxon.tree.iter">
      <class name="EmptyIterator">
         <implements name="net.sf.saxon.om.SequenceIterator"/>
         <implements                                                ↩
name="net.sf.saxon.tree.iter.ReversibleIterator"/>
         ...
         <field name="theInstance"
                type="net.sf.saxon.tree.iter.EmptyIterator"
                static="1"/>
         <method name="getInstance"
                 returns="net.sf.saxon.tree.iter.EmptyIterator"
                 static="1"/>
         <method name="nextAtomizedValue"
                 returns="net.sf.saxon.om.AtomicSequence"/>
         ...
         <class name="OfNodes">
```

```
            <extends name="net.sf.saxon.tree.iter.EmptyIterator"/>
            <implements name="net.sf.saxon.tree.iter.AxisIterator"/>
         ...
```

it indexes the class `OfNodes` with the key `net.sf.saxon.tree.iter.EmptyIterator.OfNodes`.

XSLT keys work only with nodes, not with maps and arrays, and we have no intention of changing that. Instead, the preferred approach is to construct a map that can act as an index. Often it will be appropriate for this map to be held in a global variable. How should we construct it?

In simple cases, constructing a map is easy. For example the equivalent in XPath 4.0 of a key defined with `match="employee" use="@ssn"` is a map built using `map:build(.??employee, fn{@ssn})` [4]. This case is more difficult, because with a tree of maps and arrays built from JSON, there is no ancestor axis to play with.

I experimented with several ways of constructing the index as a map. The first approach uses recursive descent template rules with tunnel parameters:

```
   <xsl:mode name="build-index" on-no-match="deep-skip"/>
   <xsl:output method="json" indent="yes"/>

   <xsl:template match="record(package, *)"
                 mode="build-index" priority="2">
      <xsl:message>Processing package {?package}</xsl:message>
      <xsl:next-match>
          <xsl:with-param name="full-name"
                          select="?package" tunnel="yes"/>
      </xsl:next-match>
   </xsl:template>

   <xsl:template match="record(class, *)" mode="build-index">
      <xsl:param name="full-name" tunnel="yes"/>
      <xsl:variable name="full-class-name"
                    select="`{$full-name}.{?class?*?name}`"/>
      <xsl:message>Processing class {$full-class-name}</       ↩
xsl:message>
      <xsl:map-entry key="$full-class-name"
                     select="{'class': ?class}"/>
      <xsl:apply-templates select="?class?*[. instance of      ↩
(record(class, *)
                                                         |    ↩
record(interface, *))]"
                           mode="#current">
          <xsl:with-param name="full-name"
                          select="$full-class-name"
                          tunnel="yes"/>
      </xsl:apply-templates>
   </xsl:template>
```

[4] `map:build`, with two arguments, returns a map in which each key-value pair contains a value from the sequence supplied in the first argument, with a corresponding key calculated using the function supplied in the second argument. The XPath 4.0 expression `fn{@ssn}` represents a function that returns the value of the `@ssn` attribute of the node supplied as the implicit function argument: in XPath 3.1 this would be written `function($node){$node/@ssn}`.

```
   <xsl:template match="record(interface, *)" mode="build-index">
      <xsl:param name="full-name" tunnel="yes"/>
      <xsl:variable name="full-class-name"
                    select="`{$full-name}.{?interface?*?name}`"/>
      <xsl:message>Processing interface {$full-class-name}</    ↩
xsl:message>
      <xsl:map-entry key="$full-class-name"
                     select="{'interface': ?interface}"/>
      <xsl:apply-templates select="?class?*[. instance of       ↩
(record(class, *)
                                                           |    ↩
record(interface, *))]"
                           mode="#current">
          <xsl:with-param name="full-name"
                          select="$full-class-name"
                          tunnel="yes"/>
      </xsl:apply-templates>
   </xsl:template>

   <xsl:template name="xsl:initial-template">
      <xsl:map>
         <xsl:apply-templates select="?digest?*"
                              mode="build-index"/>
      </xsl:map>
   </xsl:template>
```

The tunnel parameter `full-name` is used to build up the concatenated name as we descend the hierarchy; when we get to the leaf nodes, we can create a map entry using this name, so there is no need to access ancestor information. This works, but it's a lot of work to replicate a fairly simple `xsl:key` declaration. The `xsl:message` instructions are there as a reminder of how difficult I found it to get this right. The need for separate paths to handle classes and interfaces is especially irritating. They could probably be combined, but I found it was getting too complicated.

Note the use of the idiom `match="record(class, *)`. This matches any map that has an entry with the key `"class"`. A single key is often enough to identify the relevant maps uniquely.

The subtlety is that a top-level class is represented in the digest by a map that has both a `package` key and a `class` key, and both contribute to the full name of the class. By giving the `match="record(package, *)"` template rule higher priority, and then using `xsl:next-match`, we ensure that both names are added to the hierarchic name, in the right order. An inner class will have an entry that only matches the `match="record(class, *)"` template rule.

My second attempt to build the index also used recursive-descent template processing, but instead of passing tunnel parameters down with each call, it relied on the ability in a pinned tree of maps and arrays to access ancestor information. This worked, but it demonstrated no benefits over the first approach.

My third attempt used the `map:build` function, again processing a pinned tree of maps and arrays to make ancestor information. Here it is:

```
 <xsl:function name="f:fullClassName">
      <xsl:param name="c" as="(record(class, *)|record(interface,     ↩
*))"/>
      <xsl:variable name="upper"
                    select="label($c)?parent ! label(.)?parent"/>
      <xsl:variable name="prefix" select="
         if ($c instance of record(package, *))
         then $c?package
         else f:fullClassName($upper)"/>
      <xsl:sequence select="$prefix ||  '.' || $c?                    ↩
('class','interface')?*?name"/>
   </xsl:function>

   <xsl:template name="xsl:initial-template">
      <xsl:sequence select="
         map:build(
            pin(?digest)??~(record(class, *)|record(interface, *)),
            f:fullClassName#1)
         "/>
   </xsl:template>
```

I've cheated a little here, because it uses constructs that aren't yet implemented in Saxon, so I had to use workarounds to make it work. But it's only using features that are defined in the status-quo 4.0 specification.

Some observations:

- The construct `??` is a deep lookup operator: it does the same for maps and arrays as `//` does for node trees. It can be qualified by a type, so `??~record(method)` searches the entire tree for values matching the class `record(method)`. In this case we have supplied a choice type: `??~(A|B)` matches items that are instances of either `A` or `B`.
- We have called `pin()` on the tree so that each value is labelled; the label includes information about the containing (parent) map or array.
- The first argument to `map:build` selects the items to be indexed. The second computes a key value for each one. This is done by calling a user-written recursive function `f:fullClassName`.
- In this function, `$upper` navigates to the grandparent of a value. That's because the structure uses arrays of maps: to get from an inner class to its containing class, we need to go up two levels. The local name of the selected class or interface is then prefixed either with the package name (if it is a top-level class or interface), or with the full name of the containing (grandparent) class, computed by a recursive call, if it represents an inner class.

Which is preferable? Opinions will probably differ. Neither is as concise as I would like, but is the requirement frequent enough to justify custom syntax for the equivalent of an ancestor axis? With the current (very early) implementation in Saxon, both take around the same time: 500ms to 700ms to index a 5Mb digest file.

## 7. Conclusions

What have we learned from this case study? Quite a lot. We've learned about things that work well, we've learned about how to take best advantage of some of the new constructs in the language, we've generated ideas for further refinements to the language specs (some of which were implemented during the course of the study), and we've learned about areas where there is still room for further improvements.

Here's a list of some of the more important observations.

- When converting XML to JSON, we discovered the importance of achieving a mapping that i consistent not only over a large collection of instance documents, but that is also consistent over time despite the fact that tomorrow's instance documents might not have exactly the same structure as today's. We redesigned the `element-to-map` function to meet this requirement.
- The plan constructed by the `element-to-map-plan` function seems to work well on the samples we needed to convert, given a set of input documents that is sufficiently large and representative.
- We found that it's easiest to define template rules for maps if they can be written to depend only on the internal structure of the map, and not on the key used to identify the map within a larger structure. Whether this is possible depends on the design of the JSON to be tranformed. Writing match patterns of the form `match="record(real, complex)` that recognise the type of a map from the names of its fields is often a good approach. Sometimes one would also like to match on the values of a field, for example `match="record(type, *)[?type="xxx']`. It would be nice to have syntax that's less clumsy for this. There's a temptation for users to reduce it to `match=".[?type="xxx']` but this seems to lack clarity.
- XSLT often processes selected child elements or attributes by inline code within a template, and then processes the remainder using a construct such as `<xsl:apply-templates select="* except (X, Y, Z)"/>` where `X`, `Y`, `Z` are the children that have been given special treatment. The `except` operator works only on nodes, and the lookup operator `?` currently provides no similar capability to select all properties except some specifically-named ones. One option is to provide lower-priority template rules that match `X`, `Y`, and `Z` and do nothing with them.
- For this and other reasons, it is often useful to match values appearing in a tree of maps and arrays by their associated key. The syntax `match="?keyval"` has been proposed for this. The semantics, though, depend on values being labelled with their associated key, and the full complexities of this (and the usability problems that it might introduce) are not yet fully understood.
- It would be useful for the `union`, `except`, and `intersect` operators in patterns to apply to all kinds of pattern, not only patterns that match nodes. (The semantics of these operators in a pattern have already diverged in detail from their XPath semantics.)
- It would be nice to have some equivalent to `match="/"` to match the root of a tree.
- I had been concerned about how template rules should process arrays. The case study revealed no problems in this area. In most cases arrays are not processed by matching them in a template rule, but by iterating over the array in the template rule for its container.
- The current syntax for constructing maps and arrays in XSLT is rather verbose, and could be improved for many common use cases. Sometimes the right answer is to do it

in XPath: the introduction of the `fn:apply-templates` function and the `xsl:select` instruction both facilitate that. In other cases the new `xsl:record` instruction helps. An equivalent to `array:build` as an XSLT instruction has also been mooted.

- Neither `xsl:record` nor XPath map constructors make it easy to include an entry in the constructed map conditionally.
- Pinned maps and arrays make access to containing (ancestor) arrays and maps possible, but the current syntax for doing so is very clumsy.
- We've added quite a lot of functionality to introduce modifiers for lookup expressions (such as `$x?pair::y`). But this case study didn't identify any situations where they proved useful.
- When the JSON structure uses arrays of maps (which is quite common), paths such as `?x?*?y?*?z` start to appear frequently (and are very hard to debug when they select nothing). Could this be improved?

## Bibliography

[Delpratt and Lockett 2017] O'Neil Delpratt and Debbie Lockett. Distributing XSLT Processing between Client and Server. Presented at XML London 2017, June 10 - 11th, 2017. DOI: 10.14337/XMLLondon17.Lockett01. Available at https://xmllondon.com/2017/xmllondon-2017-proceedings.pdf

[Kay 2016] Michael Kay. Transforming JSON using XSLT 3.0. XML Prague 2016. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf

[Kay 2021] Michael Kay. <transpile from="Java" to="C#" via="XML" with="XSLT"/>. Markup UK, London, 2021 https://markupuk.org/2021/pdf/Markup-UK-2021-proceedings.pdf

[Kay 2022] Michael Kay. XSLT Extensions for JSON Processing. Balisage: The Markup Conference 2022, Washington, DC, August 1 - 5, 2022 https://doi.org/10.4242/BalisageVol27.Kay01

[Kay 2023] Michael Kay. Schema-Aware Conversion of XML to JSON. Presented at Balisage: The Markup Conference 2023, Washington, DC, July 31 - August 4, 2023. https://doi.org/10.4242/BalisageVol28.Kay01

# Schema Test Suite

## Building a Stronger Foundation for Schema Validation

Rebecca Bamford, Bloomsbury Publishing Plc

Francis Denton, Bloomsbury Publishing Plc

Astrea Kumaradas, Bloomsbury Publishing Plc

This paper examines the need for stronger foundations in XML schema design, particularly when working with complex content models such as DocBook, TEI, or VRA[1]. These models are based on element and attribute relationships, requiring validation scenarios that enforce structural and semantic accuracy. While significant effort goes into developing schemas to represent these complex models, their accuracy in capturing data requirements is often assumed.

To address this, the Content Architecture team at Bloomsbury Publishing have introduced a unit test-driven methodology for validating the structural integrity of Bloomsbury XML schemas. Our initial implementation of this schema test suite has proven effective in supporting schema modifications by providing reliable feedback to verify the reliability of our schemas.

## 1. Introduction

Bloomsbury Publishing's Content Strategy fundamentally centres on the question: *How do we leverage the value of content most effectively and efficiently?* The strategy becomes even more crucial when working with complex content models such as DocBook, TEI, and VRA, with their intricate relationships between elements, attributes, and content structures.

> "Fail fast, fail often."

Our approach, which encourages early and frequent testing before integration into the deployed version of a schema, is a philosophy that helps identify issues early in the workflow. Smoking out issues early on is significantly less disruptive and complex to resolve than uncovering problems at later stages in our workflows. While this approach will likely increase the number of failures reported and might seem counterintuitive, each failed test provides valuable feedback, enabling us to refine the schemas and improve publishing processes.

> "Write tests for everything."

Although we had existing frameworks in place to validate the integrity of our extensive Schematron schema, we identified a gap in the lack of structured testing for the RELAX NG and XML schemas that we work with. To address this, we decided to develop a test suite to verify and validate our customised schemas, ensuring that our content validation

[1]Visual Resources Association, a metadata schema.

standards are met now and for any future schema modifications, for every stage of our content workflow.

### 1.1. Structural Validation vs. Schematron

In XML validation, there are two primary approaches to ensure the accuracy and integrity of content:

1. Structural validation
2. Schematron

Structural validation (e.g., RELAX NG, XSD) handles the overall structure of an XML document, ensuring that elements appear in the correct order, adhere to the proper data types, and follow hierarchical relationships between elements. For instance, when a `bibliography` element is present, the `following-sibling` must be as per the following: *expected the element end-tag or element "bibliography", "glossary", "index" or "toc"*.

On the other hand, Schematron enforces a more granular, rule-based validation that can be precisely tailored to business requirements. For instance, for accessibility purposes, when `figure` elements are present, `alt` elements must be included as a child of the `mediaobject` element, or the file will raise error messages.

**Example 1. DocBook Schematron Invalid Markup**

```
<figure @xml:id="b-figure1">
      <info>
        <title>Image</title>
      </info>
      <mediaobject>
        <imageobject>
          <imagedata fileref="images/f001.jpg" format="image/jpeg"/>
        </imageobject>
      </mediaobject>
    </figure>
```

**Validation Error**: figure with @fileref value of "images/f001.jpg" must contain correct markup for descriptive text to ensure accessibility. Check element with id "b-figure1".

Both structural validation and Schematron serve different but complementary roles; combining them ensures that the XML content adheres to a required structural format and meets the customised requirements. This multi-validation approach significantly improves the reliability and validity of content throughout its lifecycle, ensuring that the high standards required in publishing are met. Additionally, we use the parsing results from industry standard validators for each flavour of schema, where any such errors should be reported by them as per usual.

### 1.2. Schema Languages

There are many different XML schema languages, and in the publishing industry, complex content models like DocBook and TEI are represented through schemas, each addressing specific content requirements:

- **DocBook**: now defined using the RELAX NG schema language, offers cleaner, more precise, and more extensible content models, making customised versions significantly easier to create than its previous DTD-based schema. [D5TDG]

- **TEI**: The Text Encoding Initiative is an XML application designed for the markup of classic literature, widely used by libraries, museums, publishers, and individual scholars to represent all textual material for online research and teaching. The guidelines define and document a markup language for representing the structural, renditional, and conceptual features of texts. [TEI]
- **VRA (Visual Resources Association) Core**: An XML-compatible metadata schema for organising works of art, namely visual resources.

For specific publishing workflows, we implement a customisation layer to our structural schemas by extending core schemas such as the DocBook Publishers schema [DBR]. This customisation is achieved with the use of an `include` statement in our customised schemas, and it allows us to build on the core standard structures whilst incorporating specific requirements and constraints that align with our publishing requirements [CDB]. Over time, the customised schemas have evolved to address the increasing complexity of the content we process; therefore, our commitment to ensuring comprehensive and accurate validation throughout the content lifecycle is vital.

## 2. Schema Test Suite

As previously identified, working with XML documents with complex content models involves enforcing structural and semantic rules through schemas. However, a critical question arises: *How can we verify the validity of the schemas to implement these structural requirements, especially as they grow in size and are tailored to suit business needs?* To address this gap, we have introduced a unit test-driven methodology for validating structural schemas.

### 2.1. Unit Testing Methodology

A unit testing methodology was best suited for the test suite design process, as this methodology allowed us to break down each schema element and treat it as an independently testable unit. This granular approach ensures that specific content models behave as expected in isolation from the schema. [UTPBQBP]

While limiting to granular, more focused tests allows for faster validation of test cases and clearer validation reports, running the full suite provides regression coverage, ensuring modifications or additions do not inadvertently break existing schema functionality.

### 2.2. Test Cases

Test cases are created in XML format, where we expect one invalid and one valid XML file per defined structure. A unique aspect of this methodology is that conventionally valid and invalid files will pass the validation process; the success of the validation process lies in the schema's ability to enforce its structural and semantic rules effectively.

#### 2.2.1. Folder Directory Hierarchy

We organise the test cases using a structured directory hierarchy to ensure consistency and clarity during testing. This organisation supports the efficient targeting of individual defined structures, for example, when testing the `role` attribute for `abstract` elements in our DocBook schema, the test case components are organised as follows:

**Example 2. Tree diagram that breaks down the file directory structure of test cases**

```
+---bloomsbury-mods-tests
¦   +---condition-attribute
¦   ¦       fail.xml
¦   ¦       pass.xml
¦   ¦
¦   +---outputformat-attribute
¦   ¦       fail.xml
¦   ¦       pass.xml
¦   ¦
¦   +---relation-attribute
¦   ¦       fail.xml
¦   ¦       pass.xml
¦   ¦
¦   +---role-attribute
¦   ¦   +---abstract
¦   ¦   ¦       fail.xml
¦   ¦   ¦       pass.xml
¦   ¦   ¦
¦   ¦   +---address
¦   ¦   ¦       fail.xml
¦   ¦   ¦       pass.xml
¦   ¦   ¦
```

**Example 3. Abstract Defined Structure**

```
db.abstract.role.attribute =
      attribute role { "blurb" | "authorAbstract" | "extract" }
        db.abstract.attlist =
          db.abstract.role.attribute
          & db.common.attributes
          & db.common.linking.attributes
```

Each test case is designed to validate against a specific defined structure, so in this instance, not only does it test a list of attributes that an `abstract` element can have, but the test also validates that the `role` attribute of `abstract` elements conforms to one of the following allowed values:

- blurb
- authorAbstract
- extract

Although the test cases can be individually targeted during testing, which should significantly reduce the processing time, they are also designed to be valid when run against the entire test suite. Executing the whole suite ensures comprehensive regression testing, flagging if new modifications to the schema have inadvertently broken existing functionality.

### 2.2.2. Templates

Test case templates are provided for consistency and accuracy for each schema, thereby reducing the likelihood of user error and simplifying test case creation. Although test cases are created manually, with the aid of test case templates, the idea of automation for future development is something to explore going forward. Automating test case creation will help accelerate and improve the accuracy of the validation process to support the ever-evolving schemas, while making the process more accessible for people with different technical backgrounds.

#### 2.2.3. Passing XML

Valid XML test cases are expected to pass schema validation since they contain structurally and semantically correct content.

For instance, the following DocBook XML file should conform to the defined structures contained in the schema being validated, where all the `role` attribute values for the `biblioset` element are permitted values:

```
<book xmlns="http://docbook.org/ns/docbook" xml:id="b-book" ↩
role="fullText">
    <info>
      <title>biblioset-role-attribute-tests</title>
      <biblioset role="publisher"/>
      <biblioset role="isbns"/>
      <biblioset role="series"/>
      <biblioset role="companionWebsite"/>
    </info>
  </book>
```

#### 2.2.4. Failing XML

Invalid XML test cases are intentionally designed to violate defined structures. They are successful when the suite correctly identifies them as invalid, demonstrating that the schema properly enforces its intended constraints.

For example, the following DocBook XML file will be flagged as a pass by the test suite, as it does not adhere to the structures defined within the schema:

```
<book xmlns="http://docbook.org/ns/docbook" xml:id="b-book" ↩
role="fullText">
    <info>
      <title>biblioset-role-attribute-tests</title>
      <biblioset role="isbn"/>
    </info>
  </book>
```

### 2.3. Implementation and Execution

#### 2.3.1. Tools and Libraries

**BaseX**: The processor for the test suite, chosen for its familiarity. Versions prior to 8.6.7 lack the required functions, so more up-to-date versions should be used.
**XQuery**: The main language used to build the test suite and incorporate BaseX's built-in XQuery functions.
**XSLT**: Primarily used to transform the test results into reports.

#### 2.3.2. Execution Logic

Once test cases have been defined, the validation suite can be used with any RELAX NG or XSD schema. The primary validation process is managed through XQuery functions, which allow users to address the following:

**Example 4. Map schemas to their corresponding test cases**

```
declare variable $sch:schemas as map(*) := map {
  "bloomsbury-mods": map {
    "schema": "..\..\content-models\DocBook\schema\bloomsbury-↩
mods.rnc",
    "test-cases": "bloomsbury-mods-tests"
  },
  "vra-strict-bloomsbury": map {
    "schema": "..\..\content-models\VRA\schema\vra-strict-↩
bloomsbury.xsd",
    "test-cases": "vra-strict-bloomsbury-tests"
  },
  "bmyTEI-tests": map {
    "schema": "..\..\content-models\TEI\schema\bmyTEI.rnc",
    "test-cases": "bmyTEI-tests"
  }
};
```

**Example 5. Target a specific schema during the test suite validation process**

```
declare function sch:options($userInput) {
  let $schema := $sch:schemas($userInput)
  return
  $schema
};
```

Additionally, BaseX's built-in XQuery validation functions are used to assess whether XML documents conform to their corresponding schemas.

**Function**: `validate:xsd-report()`

**Signature**:

```
validate:xsd-report(
              $input    as item(),
              $schema   as xs:string?  := (),
              $options  as map(*)      := {}
            ) as element(report)
```

**Summary**:

Validates the XML `$input` document against a `$schema` and returns warnings, errors and fatal errors as XML.

**Function**: `validate:rng-report()`

**Signature**:

```
validate:rng-report(
               $input    as item(),
               $schema   as xs:string,
               $compact  as xs:boolean?  := {}
             ) as element(report)
```

**Summary**:

Validates the XML `$input` document against a `$schema`, using the XML or `$compact` notation, and returns warnings, errors and fatal errors as XML.

### 2.3.3. Reporting

Test results are captured in human-readable, queryable reports, which are generated and exported in multiple formats. Built-in BaseX reporting includes the following:

#### Trace logs

Immediate user reporting is provided during the suite's execution and once it is complete. The `trace()` function offers logs during the validation process, which can be useful for debugging. The function returns the directory path for all items being tested in that execution run.

**Example 6. Trace log**

"Validating: C:\Resources2\xquery\schema\tests\bloomsbury-mods-tests\role-attribute\book\book-role-attribute-fullText\fail.xml"

"Validating: C:\Resources2\xquery\schema\tests\bloomsbury-mods-tests\role-attribute\book\book-role-attribute-fullText\pass.xml"

"Validating: C:\Resources2\xquery\schema\tests\bloomsbury-mods-tests\role-attribute\book\book-role-attribute-pdfOnly\fail.xml"

"Validating: C:\Resources2\xquery\schema\tests\bloomsbury-mods-tests\role-attribute\book\book-role-attribute-pdfOnly\pass.xml"

#### BaseX Result Window

This summarises the test results in a user-friendly manner, displaying the number of test cases that pass and fail.

**Example 7. Valid Test Cases**

***0 TESTS FAILED***

***94 TESTS PASSED***

Further reporting is also provided in the scenario of a test case which fails validation. The reports will provide the following:

1. A direct path to the XML that failed the test suite validation.
2. Overview of the result.
3. Descriptive user error messages for easy debugging.

**Example 8. Invalid Test Cases**

***1 TESTS FAILED***

***93 TESTS PASSED***

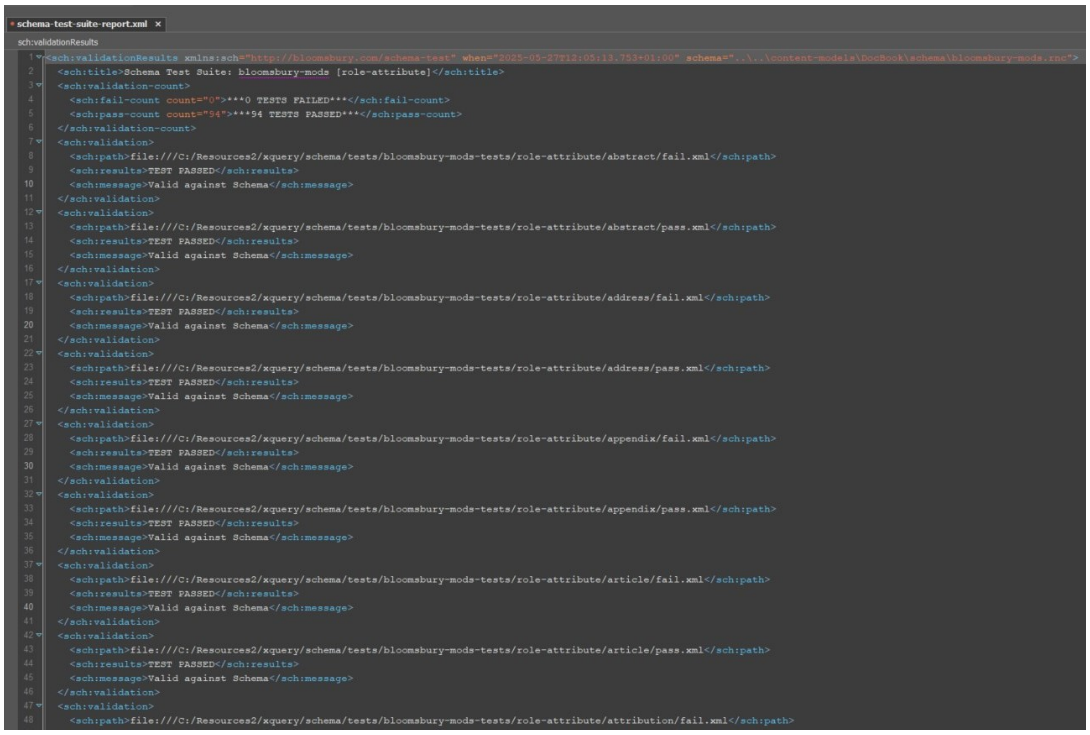
```
<sch:validation xmlns:sch="http://bloomsbury.com/schema-test">
  <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-     ↩
mods-tests/role-attribute/book/book-role-attribute-fullText/pass.xml</↩
```

```
sch:path>
  <sch:results>***TEST FAILED***</sch:results>
  <sch:message>
    <report>
      <status>invalid</status>
      <message level="Error" line="1" column="115"                      ↩
url="file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/  ↩
role-attribute/book/book-role-attribute-fullText/pass.xml">value of    ↩
attribute "role" is invalid; must be equal to "fullText" or            ↩
"pdfOnly"</message>
    </report>
  </sch:message>
</sch:validation>
```

```
<sch:validation xmlns:sch="http://bloomsbury.com/schema-test">
  <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-      ↩
mods-tests/role-attribute/book/book-role-attribute-fullText/fail.xml</↩
sch:path>
  <sch:results>***TEST FAILED***</sch:results>
  <sch:message>fail.xml valid against Schema which is incorrect</      ↩
sch:message>
</sch:validation>
```

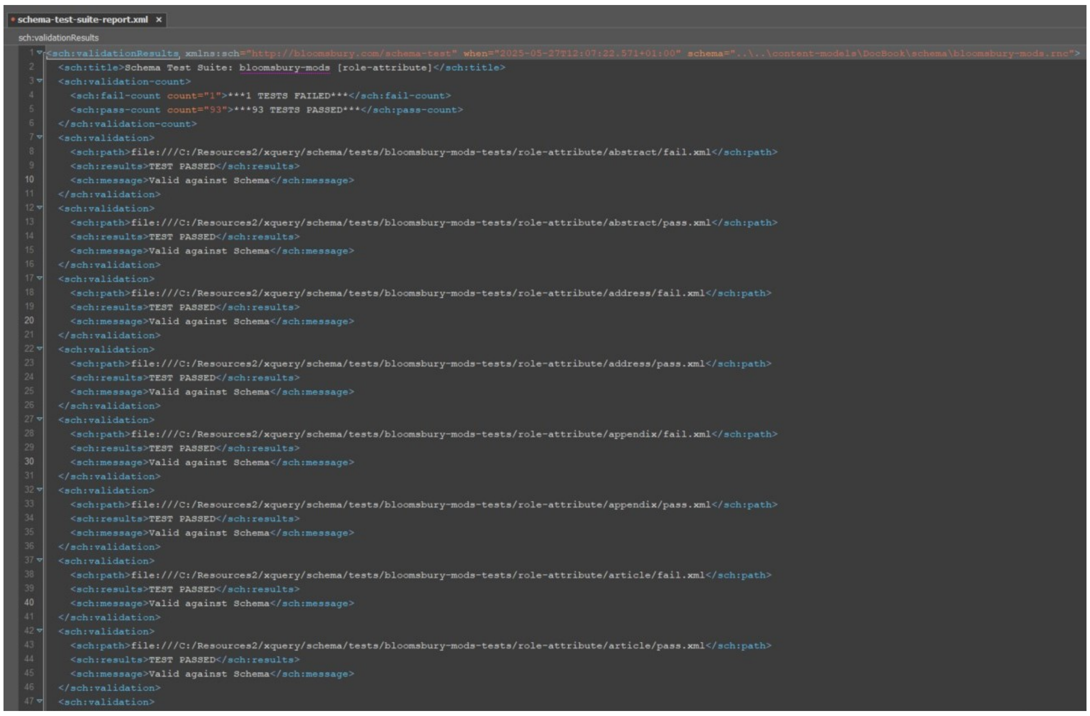
### XML Reporting

Figure 1. XML file of schema report where test cases pass validation.



```
<sch:validationResults xmlns:sch="http://bloomsbury.com/schema-test" when="2025-05-27T12:05:13.753+01:00" schema="..\..\content-models\DocBook\schema\bloomsbury-mods.rnc">
  <sch:title>Schema Test Suite: bloomsbury-mods [role-attribute]</sch:title>
  <sch:validation-count>
    <sch:fail-count count="0">***0 TESTS FAILED***</sch:fail-count>
    <sch:pass-count count="94">***94 TESTS PASSED***</sch:pass-count>
  </sch:validation-count>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/abstract/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/abstract/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/address/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/address/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/appendix/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/appendix/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/article/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/article/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/attribution/fail.xml</sch:path>
```

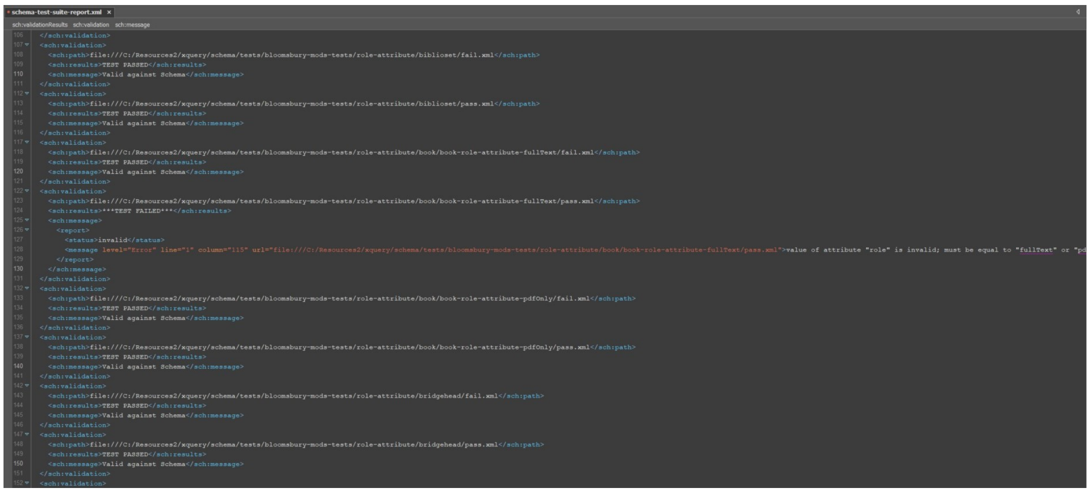
Figure 2. XML file of schema report where test cases fail validation.



schema-test-suite-report.xml ×

sch:validationResults

```
<sch:validationResults xmlns:sch="http://bloomsbury.com/schema-test" when="2025-05-27T12:07:22.571+01:00" schema="..\..\content-models\DocBook\schema\bloomsbury-mods.rnc">
  <sch:title>Schema Test Suite: bloomsbury-mods [role-attribute]</sch:title>
  <sch:validation-count>
    <sch:fail-count count="1">***1 TESTS FAILED***</sch:fail-count>
    <sch:pass-count count="93">***93 TESTS PASSED***</sch:pass-count>
  </sch:validation-count>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/abstract/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/abstract/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/address/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/address/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/appendix/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/appendix/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/article/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/article/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
```



schema-test-suite-report.xml ×

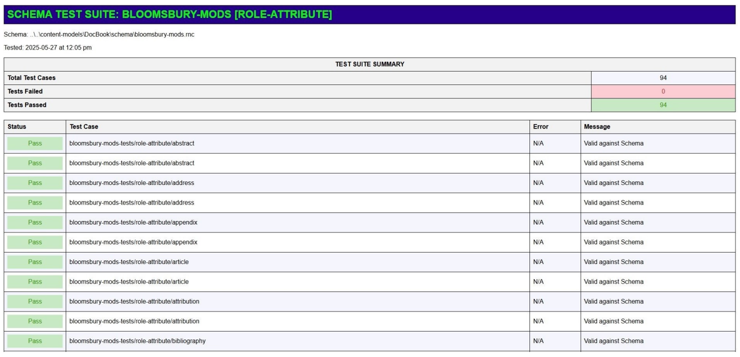
sch:validationResults sch:validation sch:message

```
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/biblioset/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/biblioset/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-fullText/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-fullText/pass.xml</sch:path>
    <sch:results>***TEST FAILED***</sch:results>
    <sch:message>
      <report>
        <status>invalid</status>
        <message level="Error" line="1" column="115" url="file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-fullText/pass.xml">value of attribute "role" is invalid; must be equal to "fullText" or "pd
      </report>
    </sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-pdfOnly/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-pdfOnly/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/bridgehead/fail.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
    <sch:path>file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/bridgehead/pass.xml</sch:path>
    <sch:results>TEST PASSED</sch:results>
    <sch:message>Valid against Schema</sch:message>
  </sch:validation>
  <sch:validation>
```

## HTML Reporting

HTML reports are generated through the transformation of XML reports via XSLT.

Figure 3. An HTML file of the schema report where test cases pass validation.



SCHEMA TEST SUITE: BLOOMSBURY-MODS [ROLE-ATTRIBUTE]

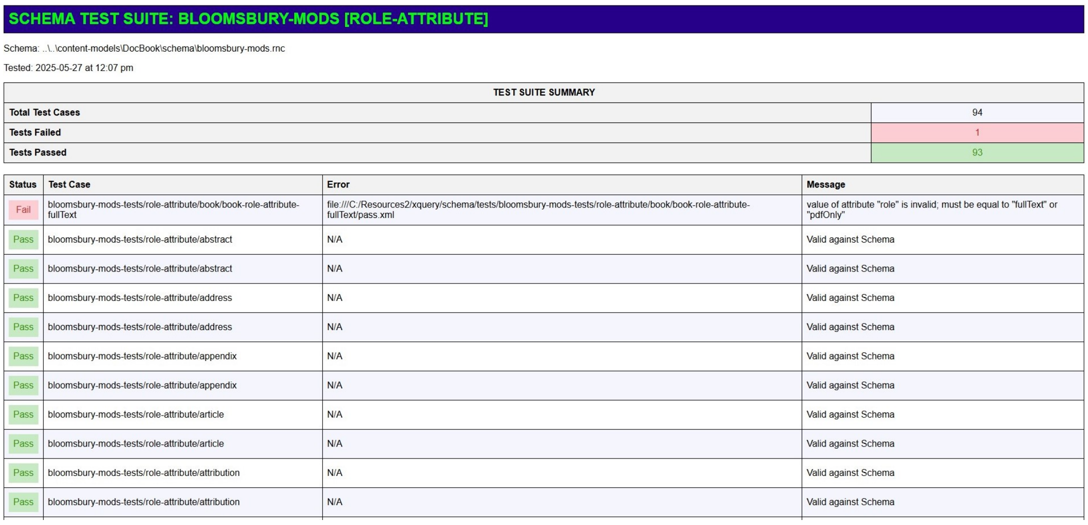
Schema: ..\..\content-models\DocBook\schema\bloomsbury-mods.rnc

Tested: 2025-05-27 at 12:05 pm

| TEST SUITE SUMMARY | |
|---|---|
| Total Test Cases | 94 |
| Tests Failed | 0 |
| Tests Passed | 94 |

| Status | Test Case | Error | Message |
|---|---|---|---|
| Pass | bloomsbury-mods-tests/role-attribute/abstract | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/abstract | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/address | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/address | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/appendix | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/appendix | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/article | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/article | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/attribution | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/attribution | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/bibliography | N/A | Valid against Schema |

Figure 4. An HTML file of the schema report where test cases fail validation.



SCHEMA TEST SUITE: BLOOMSBURY-MODS [ROLE-ATTRIBUTE]

Schema: ..\..\content-models\DocBook\schema\bloomsbury-mods.rnc

Tested: 2025-05-27 at 12:07 pm

| TEST SUITE SUMMARY | |
|---|---|
| **Total Test Cases** | 94 |
| **Tests Failed** | 1 |
| **Tests Passed** | 93 |

| Status | Test Case | Error | Message |
|---|---|---|---|
| Fail | bloomsbury-mods-tests/role-attribute/book/book-role-attribute-fullText | file:///C:/Resources2/xquery/schema/tests/bloomsbury-mods-tests/role-attribute/book/book-role-attribute-fullText/pass.xml | value of attribute "role" is invalid; must be equal to "fullText" or "pdfOnly" |
| Pass | bloomsbury-mods-tests/role-attribute/abstract | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/abstract | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/address | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/address | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/appendix | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/appendix | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/article | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/article | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/attribution | N/A | Valid against Schema |
| Pass | bloomsbury-mods-tests/role-attribute/attribution | N/A | Valid against Schema |

## 3. Conclusion

We have established a foundation for sustainable and reliable XML schema development by applying a structured, unit-test-based approach to schema validation, enhancing confidence in validity. It supports long-term quality assurance, and the principles and practices outlined here are applicable across multidisciplinary teams and domains, which can serve as a foundation for stronger schema validation.

### 3.1. Continuous Development

As mentioned, future work will explore automating test case creation to improve the validation process further. Additionally, the archiving of reports, which could aid debugging and provide a valuable audit trail of schema modifications, will be explored.

### 3.2. Integration

Validation must be embedded into the schema development lifecycle to ensure and maintain high schema quality. Our approach integrates validation at every stage by:

1. Regular regression testing - running the entire schema during each cycle, not just a subset of the schema.

2. Continuous improvement - automating test case creation and archiving reports.

3. Expansion - incrementally developing test cases to reflect the full breadth of the schemas, ensuring complete rule coverage.

### 3.3. Accessibility

The test suite has been intentionally designed for accessibility across varying technical backgrounds, so users only require a basic understanding of XML to run and interpret test results. Embedding validation into the wider development workflow encourages shared ownership of schema quality and accuracy, and by lowering the technical barrier to make schema validation more accessible, it can be adopted more broadly within multidisciplinary teams.

## Bibliography

[D5TDG] Walsh, N. and Hamilton, R.L. (2010). DocBook 5: The Definitive Guide. O’Reilly Media, Inc

[TEI] tei-c.org. (n.d.). TEI: Text Encoding Initiative. [online] Available at: https://tei-c.org/

[VRAC] Visual Resources Association (VRA) Core. (n.d.). Available at: https://elizabethbradshaw.wordpress.com/wp-content/uploads/2015/12/final_paper_pdf.pdf

[DBR] Docbook.org. (2025). DocBook Release. [online] Available at: https://docbook.org/xml/5.1cr1/

[CDB] Docbook.org. (2020). Customizing DocBook. [online] Available at: https://tdg.docbook.org/tdg/5.1/ch05#ch05-layers

[UTPBQBP] CodeFresh (n.d.). Unit Testing: Principles, Benefits & 6 Quick Best Practices. [online] Codefresh. Available at: https://codefresh.io/learn/unit-testing/

We thank our fellow Bloomsbury Content Analysts and Content Architecture colleagues who contributed to the testing infrastructure and schemas that made this project possible.

# Design and Performance of a Corpus Scanner

Liam Quin, Delightful Computing

People working with large collections of XML documents often need to know specific characteristics of the documents in the collection in aggregate. For example, an attribute value that only occurs once in a million documents might warrant investigation; an element that was expected but that does not occur anywhere might similarly suggest a problem. People designing transformations or style sheets might find it useful to handle the most commonly occurring elements first.

FreqX is an XSLT-based tool that summarizes the various elements, attributes, attribute values, and other details in a collection of XML documents. It can produce several different report formats, including an HTML Web page, a CSV file for a spreadsheet, and of course XML data. It has been run on collections containing tens of thousands of documents, running into tens of gigabytes of XML.

Unfortunately, early versions of the tool used large amounts of memory—several times more memory than the actual scanned documents occupied. This made the tool unsuitable for one of its design goals.

Recently, the FreqX tool has been improved so that it runs more quickly and uses much less memory.

This paper describes some of the design decisions that were made both in the creation of FreqX and in the subsequent revision, and also the process of making the tool support large amounts of data.

The tool is written in XSLT 3, and makes use of a number of XPath and XSLT features introduced in that version. Some of these are discussed in the paper. FreqX is publicly available, including full source code, with original development funded by Mulberry Technologies. Suggestions for additional features, as well as reports of problems, are welcomed: the tool is actively maintained.

The result of the improvements was a reduction in memory usage from over sixty gigabytes to less than three gigabytes when processing the Early English Books Online corpus of some fifty-three thousand TEI-based XML documents, and a reduction in time from almost six hours before a crash down to between thirty and forty-five minutes with successful output, running in both cases with Saxon 9 on a decade-old computer.

## 1. Introduction

FreqX is a tool that produces reports about elements and attributes found in a body or *corpus* of XML documents. Some uses of FreqX have included:

- Researching elements that could be dropped from a new version of a vocabulary;

- Investigating whether there were values of *role* or *class* attributes that were used frequently enough to suggest a new element to represent the concept;
- Investigating a large body of documents (thousands or tens of thousands) as part of maintaining or writing transformations or other software;
- Producing pretty reports for conference papers or client reports (the importance of this should not be underestimated).

This paper discusses some of the challenges that one encounter when writing such a tool, and some of the (often arbitrary) design decisions taken.

Some of the challenges included:

- Supporting multiple ways to specify which documents to process;
- Handling documents that has parse errors in them;
- Running in a reasonable time;
- Not running out of memory;
- Coping with documents requiring different DTD files for the same PUBLIC identifier;

Initial FreqX development was sponsored by Mulberry Technologies. They wanted the tool written in XSLT since that was their primary language, and since it's also the author's, this was a good fit.

Subsequent development was funded by Delightful Computing, with help from Gerrit Imsieke of Le-Tex Publishing and others.

## 2. Tool Requirements and Features

This section gives a brief overview of the features. Its purpose is to give the reader enough context to follow the rest of the paper. Further features, that were introduced to overcome limitations or problems, are discussed in later sections This paper is not intended to be an introduction to the tool itself, but to describe development of the tool.

### 2.1. Easy to configure and run

It must be easier in most cases to run FreqX than to write ad hoc queries or to write something else.

An external XML configuration file can be used, so that a run can be duplicated or refined.

FreqX can be run using a batch shell script (supplied) or from an environment such as Oxygen XML Editor.

### 2.2. Convenient to provide inputs

Just as it must be easy to run and configure the tool, it must also be easy to give FreqX a corpus of documents.

Currently, FreqX can read an XML document listing files to process, which can be in Saxon collection format or Oxygen project format; it can explore a folder recursively; it can use Saxon collection arguments; it may also be possible to configure it using a suitable resolver to explore BaseX database collections, although this has not been tested.

### 2.3. Produce multiple forms of report

Currently FreqX can produce HTML, CSV, and an XML format that it can also read for combining runs. Other report formats can be added. The author has also explored a JSON report suitable for use with various `d3.js` visualizations.

### 2.4. Combine multiple runs into a single report

As mentioned in the previous section, FreqX can read its own output. As a result it is possible to run FreqX on multiple sets of documents, possibly with different options such as different XML catalogue files, and produce combined reports.

### 2.5. Robust against parse errors

A non-well-formed or invalid document must not cause FreqX to crash. Instead, a summary of parse errors shall be produced.

Large corpora are likely to have problems. Sometimes they are collections of different sorts of documents, or may include intermediate formats or even a sub-folder of documents known by its creators to be erroneous. FreqX produces a list of errors in the HTML report, limited by default to the first fifty erroneous documents.

### 2.6. Extensible and Maintainable

It must be straight forward to add a new report. In addition, the XSLT should be documented internally and readable.

This requirement is subjective, but the places to edit to add a new report are in options, help, and report functions.

## 3. Implementation

An early version of FreqX used a recursive template to process one input file at a time. This quickly ran out of memory, and was replaced with the XSLT 3 `xsl:iterate` instruction. This still ran out of memory, and we shall return to this topic in a later section, but it handled many more documents.

The first strategy was to produce an XPath map structure from each input document, containing a list of all distinct elements found, and all attributes and all attribute values, and then to merge the maps at each iteration.

This strategy turned out to use too much memory. A rewrite using elements instead of maps to represent the data was faster and used less memory.

It’s entirely likely that the problem was actually not that maps were slower than elements in Saxon 9, but that it’s easy to include a fragment of an input document in a map, and, unlike when it’s copied into element content, the fragment is a live node in a document tree, and hence the entire document must be kept in memory to support possible XPath navigation away from the node. This is especially easy to do by mistake with attribute nodes.

A way to check for this can be to run a stylesheet in XSLT streaming mode, because any template that then tries to return data from an input stream will raise an error. Nodes must be “grounded” instead, for example, using the XPath 3 `fn:copy-of()` function. However, converting a transformation to work correctly in streaming mode is generally non-trivial.

The current implementation makes an `e` element for each element in the input, and an `at` element for each attribute seen. These are then turned into distinct values at the end of processing each input file, and converted into `count` elements that represent totals seen so far.

The process of combining `at` elements repersenting the set of attribute values seen so far turns out to be slow. It might be that a map would be faster than the current method; profiling showed it was slow, and a workaround was to merge attribute values after every hundred input documents, and at the end of processing:

```
<xsl:with-param name="attribute-values-seen"
  select="
    if ((position() ge last() - 1) or (position() mod 100) eq 99)
    then
      dc:merge-attribute-values(
        $attribute-values-seen,
        $this-data-set[local-name() = 'at']
      )
    else (
      $attribute-values-seen,
      dc:merge-attribute-values(
        (),
        $this-data-set[local-name() = 'at']
      )
    )
  " />
```

The *merge-attributes-seen()* function makes a *count* element for each distinct attribute name/value combination; there can easily be millions of these, and as the sequence of pairs already discovered grows large, the process slows down considerably. So there is a trade-off between extra memory for duplicated attribute/value *count* elements and CPU time. Running *merge-attributes-seen()* only at the end of processing can use a lot of memory, but running it for every document is slow.

Although the same consideration applies to element names and to attribute names, there are generally many fewer of these.

## 4. Memory Usage and Speed

Gerrit Imsieke ran Saxon in profiling mode and discovered that, as mentioned in the previous section, merging the list of seen attributes was very slow. *Not* merging them saved a lot of time but produced wrong answers.

This is when the author started to suspect that the entire input was being kept in memory. The Saxon *uri-collection()* function was being called with a *stable=no* parameter, which the documentation seemed to suggest would mean the documents would not need to be kept in memory.

It appears, based on testing, that in fact *stable=no* simply means that calling *uri-collection()* multiple times in the same run might not always return the same set of document URIs (filenames). It does *not* mean that the documents themselves are not guaranteed to be stable, and hence does *not* mean the documents are not kept in memory.

In the end what worked was processing each input document in an external stylesheet, called using the XPath 3 *fn:transform()* function. Since each invocation of XSLT was separate, it seemed memory was not retained between them, and FreqX ran much faster.

```
xsl:sequence select="transform(
  map {
    'stylesheet-location' : $process-file-xsl,
    'initial-template' : QName((), 'initial-template'),
    'stylesheet-params' : map {
        QName('http://www.delightfulcomputing.com/', 'freqx-control-  ↩
doc') : $freqx-control-doc,
        QName('http://www.delightfulcomputing.com/', 'freqx-input-    ↩
uri') : $this?name
    }
  }
)?output/*" />
```

Performance can to some extent be measured using the builtin profiling in Saxon; an alternative is to transform the XSLT style sheet to add `xsl:message` instructions at the start and end of each template of interest, and then to analyze timestamps on the log file.

Since XPath functions are deterministic, functions such as *fn:current-time()* always return the same value within a single XSLT episode. Therefore the time for profiling must be reported either with a Java native method call, or by using an external tool such as the combination of *ts* and *unbuffer* from the Linux more-utils package. The *ts* command adds timestamps to each line of input. However, program standard output is buffered for efficiency and is delivered in clumps when the buffer is full. The *unbuffer* command can be used to prevent that buffering and get accurate timestamps.

Timings can also be obtained by separate runs of the external stylesheet that would be called using *fn:transform()*, and memory can be measured, for example on Linux or Unix systems with */usr/bin/time* (*time* without the path is a built-in in many shells, including *bash*, that gives less information).

People running timings need to keep overall system activity in mind, as well as overall system memory usage.

In the case of FreqX, memory rose to over sixty gigabytes on a test collection, and it became clear it was keeping all of the documents in memory.

Changing FreqX to use fn:transform() on the result of doc($filename) did not help.

Passing $filename as a parameter to the external stylesheet *did* help: runtime was reduced dramatically, as was memory usage.

A further refinement was to keep namespace URIs as integer keys into a map instead of strings, but the additional complexity of passing that back from the external stylesheet did not seem justified; the author may return to this in the future.

## 5. Parsing Errors

Saxon extends the *fn:uri-collection()* function to be able to descend recursively into a directory (folder) structure and return all files found whose names match a pattern. FreqX uses this as one of its mechanisms to obtain a lit of files to analyze. But not all input files will always be DTD-valid, and some may not even be well-formed XML at all, regardless of file name.

FreqX wraps each call to open a file (using the *fn:doc()* function) inside xsl:try and xsl:catch, so that parse errors do not terminate processing. It then records the errors. Since there may be a great many errors, by default only the first fifty are recorded, and are included in an expandable *summary/details* section in the HTML report.

The most common parse error is not finding a DTD. The FreqX wrapper script takes optional filenames for a Java catalog resolver class and an XML catalog file. However, Saxon only supports using one such catalog file in any given run, and a large corpus might well contain documents using different incompatible versions of a DTD identified by the same identifier, whether SYSTEM or PUBLIC.

FreqX can be run multiple times, and the counts combined. A future version might be able to process only the failed documents from a previous run, so that one can more easily combine results from runs with different XML catalog files.

## 6. Extensibility

FreqX uses an array of maps to represent information about available reports. In the following code listing, the `dc:v()` function produces a string, empty in the case of an empty sequence. This is needed as otherwise an empty sequence in a comma-separated sequence would be discarded, and the columns in the CSV files would not align. This is not required for numbers, as zero values are not discarded. An alternative design might have used an array, as these can contain empty sequences. With XSLT 4, a record type would provide increased type safety.

```
<xsl:variable name="csv-makers" as="map(*)"
  select="
      (: This data structure drives the various different
       : comma-separated-value (CSV) reports.
       :)
     map {
      'elements' : map {
        'what' : function($counts) { $counts/elements/* },
        'headings' : 'Element,NS,Nocc,NDocs',
        'attributes' : function($count as element(count)) {
          (
            dc:v($count/@name),
            dc:v($count/@ns),
            xs:string($count),
            dc:v($count/@ndocs)
          )
        }
     },
     'element-parents' : map {
       'what' : function($counts) { $counts/elements-parents/* },
       'headings' : 'Element,NS,Parent,Parent NS,Nocc,NDocs',
       'attributes' : function($count as element(count)) {
         (
            dc:v($count/@name),
            dc:v($count/@ns),
            dc:v($count/@parent-name),
            dc:v($count/@parent-ns),
            xs:string($count),
            dc:v($count/@ndocs)
         )
       }
     },
```

The listing is incomplete: there are more entries in the actual XSLT file for FreqX, and of course a new CSV report can be added by inserting a new entry into the map. Entirely new formats, such as JSON, require a separate new template, but the hardest part is deciding how to represent the information in the report.

The report generator will apply the *attributes* function to each count element in turn, receive a sequence of strings in return, and make them into one item of the report: one comma-separated line, for example. The function examines the attributes of *count* elements to obtain information such as names, namespaces, counts.

This architecture means that the representation of final counts as count elements can be changed with only moderate effort, and the representation of observations is entirely self-contained in the document scanner. As a result it would be possible to experiment again with maps instead of elements, for example.

## 7. Future Work

Although FreqX is much faster than it used to be, it could probably be faster.

Using *fn:transform()* might let multiple XML catalog files be used. If not, the author has written an XSLT-based DTD parser that could possibly be integrated and used (at the expense of speed) where needed. It is not, at the time of writing this paper, possibly to supply a catalog resolver to the XPath *fn:doc()* function, unfortunately. FreqX does, however support combining results from a previous run with a different XML catalogue file in use, which can mitigate this problem.

Additional output formats and additional visualizations may be helpful. A JSON report is in the works, along with a treemap visualization based on *d3.js*.

## 8. Conclusion

Using fn:transform() on each input file helped to control memory usage, and proved more effective than several other strategies tried. The input filename is passed to the external template, not the parsed tree.

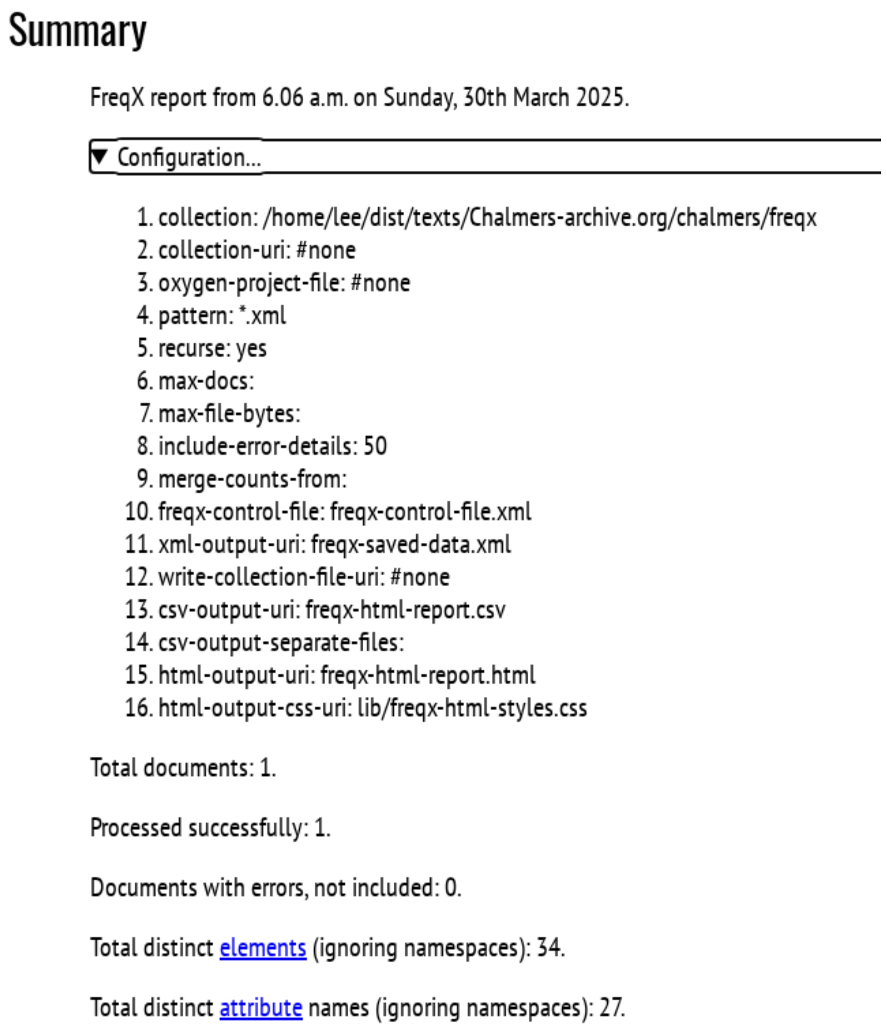
FreqX provides a useful overview of a corpus, and now runs in a reasonable time.

## A. Samples

This section includes some screenshots from an HTML report produced by FreqX to give an idea of the results. The report is interactive, so some of the screenshots show the result of a user clicking on (activating) an element or attribute name.

### Figure A.1. Report Summary

The Configuration section is collapsed by default, but is expended here to show the options.



## Summary

FreqX report from 6.06 a.m. on Sunday, 30th March 2025.

▼ Configuration...

1. collection: /home/lee/dist/texts/Chalmers-archive.org/chalmers/freqx
2. collection-uri: #none
3. oxygen-project-file: #none
4. pattern: *.xml
5. recurse: yes
6. max-docs:
7. max-file-bytes:
8. include-error-details: 50
9. merge-counts-from:
10. freqx-control-file: freqx-control-file.xml
11. xml-output-uri: freqx-saved-data.xml
12. write-collection-file-uri: #none
13. csv-output-uri: freqx-html-report.csv
14. csv-output-separate-files:
15. html-output-uri: freqx-html-report.html
16. html-output-css-uri: lib/freqx-html-styles.css

Total documents: 1.

Processed successfully: 1.

Documents with errors, not included: 0.

Total distinct elements (ignoring namespaces): 34.

Total distinct attribute names (ignoring namespaces): 27.

### Figure A.2. Element Frequency

This report has one line for each element name seen, sorted by frequency across all input documents. The report starts (after configuration and errors) with top-level elements seen, partly because any surprises here need to be attended to forthwith. This report was run on a single fifty-megabyte XML document.

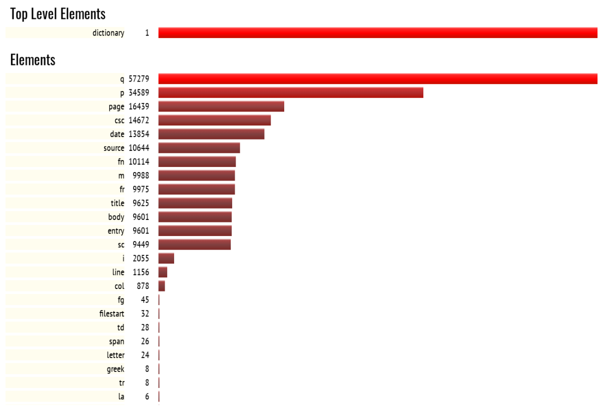
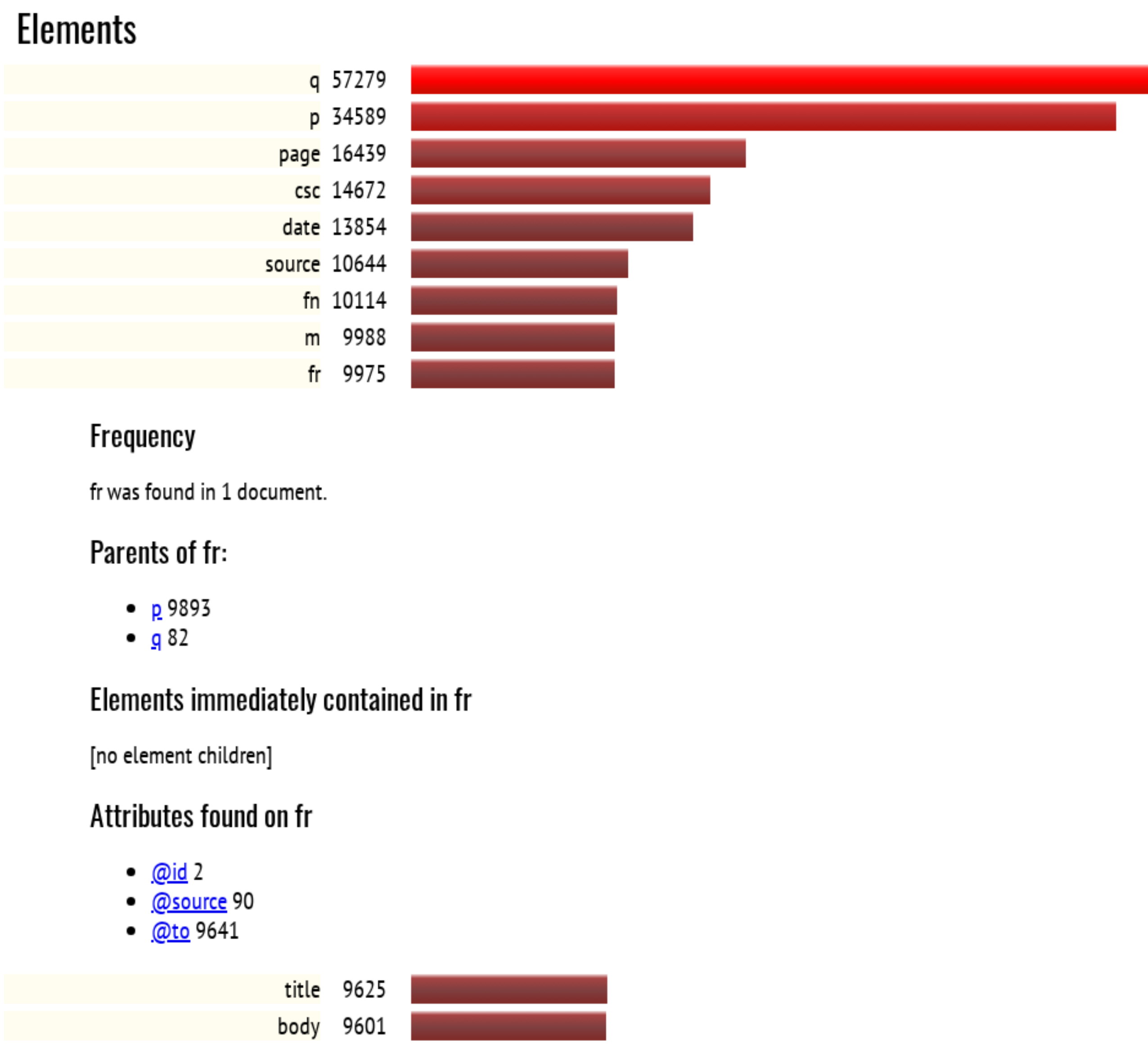


**Figure A.3. Element Details**

In this figure the user has clicked on the *fr* (footnote reference) element. It has expanded to show *fr* was found in one document (not surprising as only one document was analyzed in this run), that it occurred inside *p* and *q* (quote) elements, and that 9641 of the *fr* elements had an *fr* attribute. In the dictionary an *fr* element without a *to* attribute usually represents an error.



Elements

| | |
|---|---|
| q | 57279 |
| p | 34589 |
| page | 16439 |
| csc | 14672 |
| date | 13854 |
| source | 10644 |
| fn | 10114 |
| m | 9988 |
| fr | 9975 |

Frequency

fr was found in 1 document.

Parents of fr:

- p 9893
- q 82

Elements immediately contained in fr

[no element children]

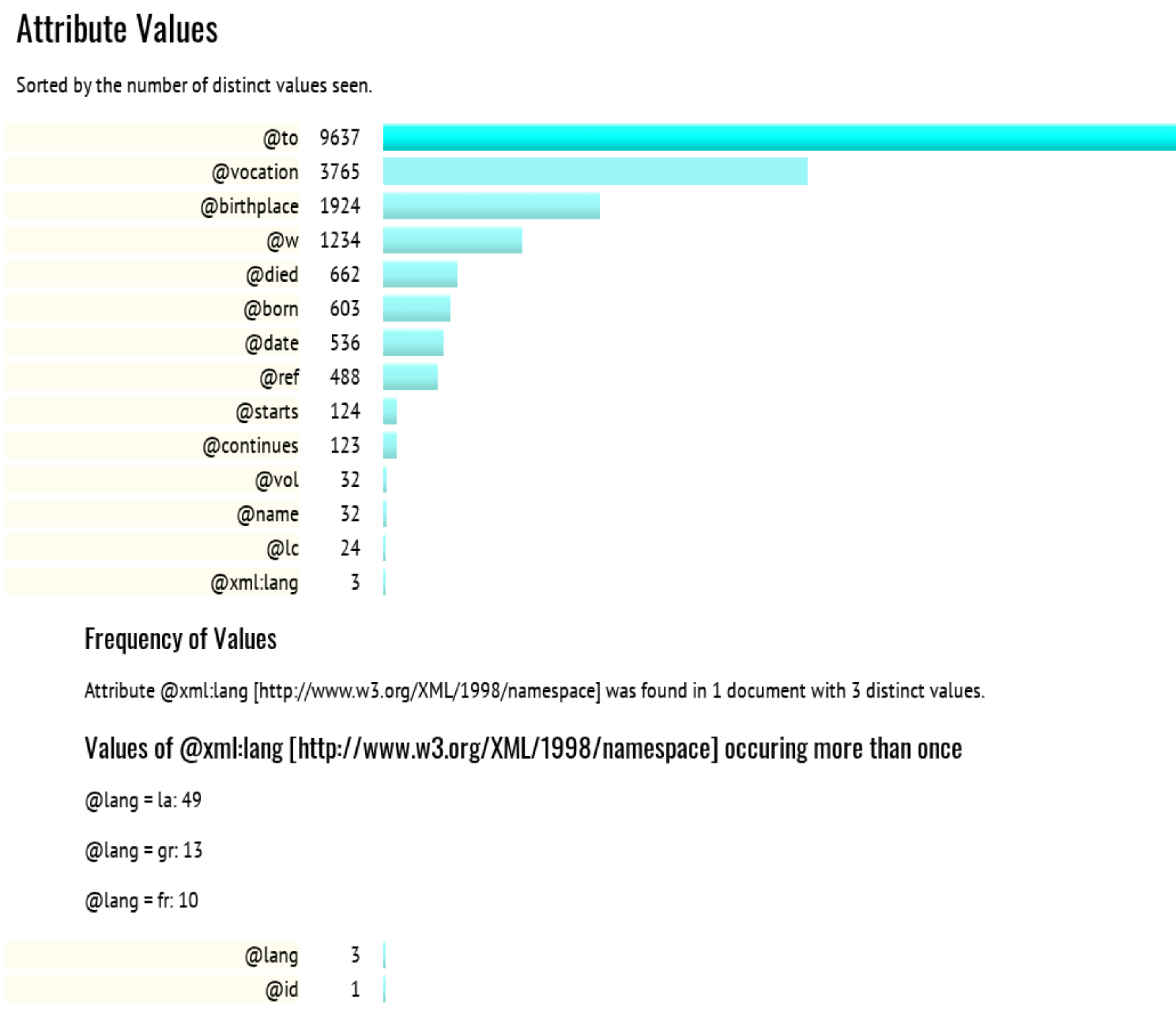
Attributes found on fr

- @id 2
- @source 90
- @to 9641

| | |
|---|---|
| title | 9625 |
| body | 9601 |

**Figure A.4. Attribute Values**

The part of the report showing attributes is not included in this paper because it is similar to the part showing elements. This figure shows the final report from FreqX, the distinct values of attributes. The entry for *xml:lang* has been expanded.



## Attribute Values

Sorted by the number of distinct values seen.

| | |
|---|---|
| @to | 9637 |
| @vocation | 3765 |
| @birthplace | 1924 |
| @w | 1234 |
| @died | 662 |
| @born | 603 |
| @date | 536 |
| @ref | 488 |
| @starts | 124 |
| @continues | 123 |
| @vol | 32 |
| @name | 32 |
| @lc | 24 |
| @xml:lang | 3 |

### Frequency of Values

Attribute @xml:lang [http://www.w3.org/XML/1998/namespace] was found in 1 document with 3 distinct values.

### Values of @xml:lang [http://www.w3.org/XML/1998/namespace] occuring more than once

@lang = la: 49

@lang = gr: 13

@lang = fr: 10

| | |
|---|---|
| @lang | 3 |
| @id | 1 |

# Surfing the web with XProc

Norm Tovey-Walsh

XProc 3.1 comes with many great features for surfing the web. The p:document instruction will load XML, HTML, or JSON documents. The p:http-request step allows a pipeline author to interact with web services. It supports many HTTP methods, query parameters, content negotiation, and single and multipart request and response bodies.

But what about web applications? Web pages that rely on client-side processing, with XSLT using SaxonJS, for example, or just plain old Javascript, present a special challenge. What can we do about that?

## 1. Introduction

Back in January, on the xproc-dev mailing list, Andy Carver asked about [https://lists.w3.org/Archives/Public/xproc-dev/2025Jan/0063.html] “grabbing web pages”. In principle, this is easy. XProc 3.1, like the web itself, supports an extensible set of document types identified by content type. All XProc implementations support XML, HTML, JSON, text, and binary types. Implementors may support additional content types; XML Calabash supports YAML and a variety of RDF content types, for example.

This means that an XProc pipeline can load those documents in a completely straightforward way. Consider this pipeline:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                name="main" version="3.1">
  <p:output port="result"/>
  <p:option name="uri"/>

  <p:load href="{$uri}"/>

</p:declare-step>
```

This pipeline simply loads a document and serializes it. For example:

```
$ bin/xmlcalabash.sh pipelines/document.xpl uri=https://                ↩
testdata.xmlcalabash.com/index.xml
```

Returns the XML document:

```


<?xml version="1.0" encoding="UTF-8"?><html xmlns="http://www.w3.org/  ↩
```

```
1999/xhtml">
<head>
<meta charset="utf-8"/>
…
</body>
</html>
```

And:

```
$ bin/xmlcalabash.sh pipelines/document.xpl uri=https://                  ↩
testdata.xmlcalabash.com/index.html
```

Returns an HTML document:

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta http-equiv="Content-Type" content="text/html;                  ↩
charset=UTF-8">
…
      </body>
</html>
```

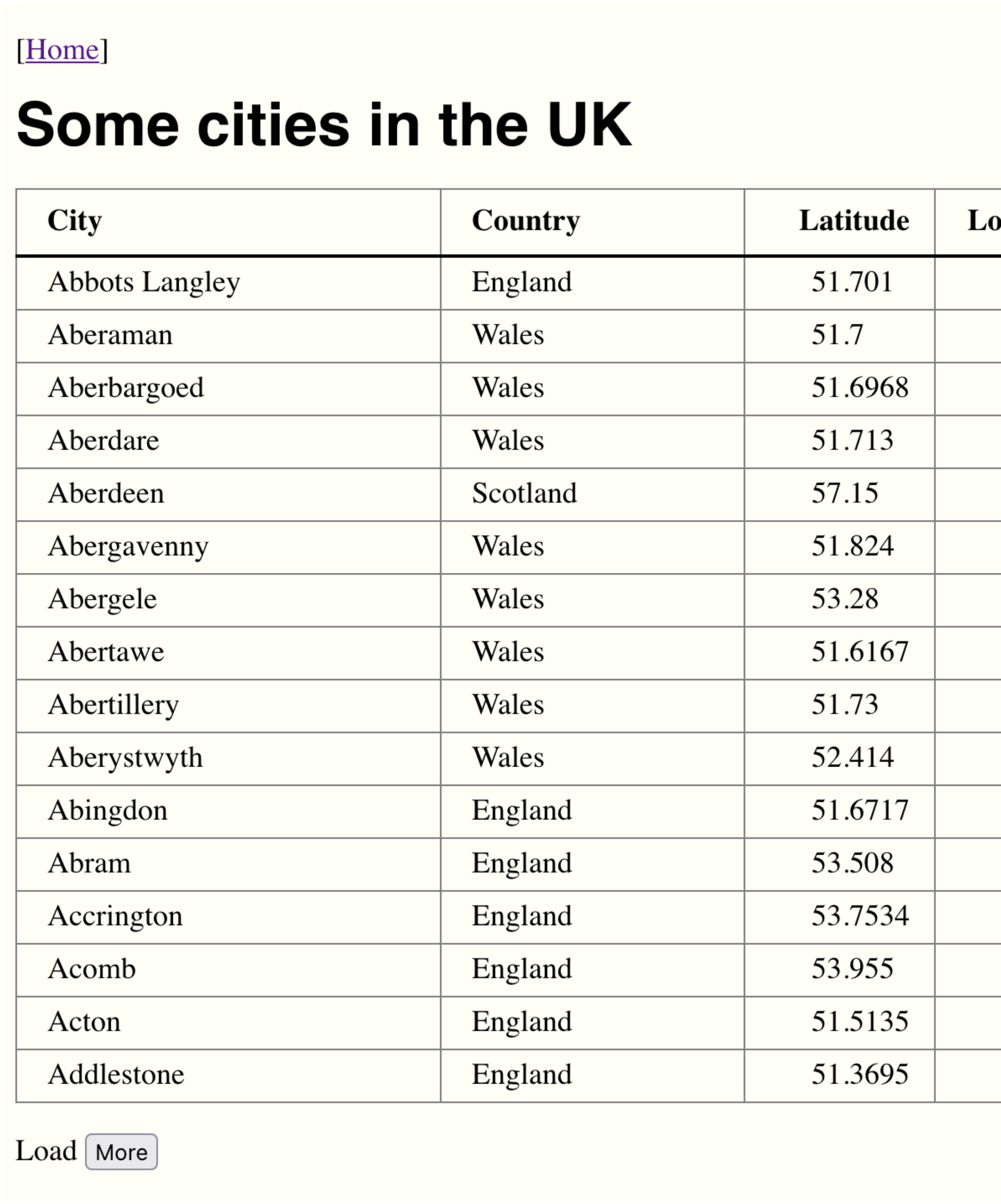
The document served on `https://testdata.xmlcalabash.com/index.html` begins like this:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta charset="utf-8">
<title>Index</title>
<link rel="stylesheet" href="style.css">
</head>
…
```

XProc has no trouble loading this “non-XML” document because it is served with an HTML content type. So far, everything is looking pretty good for surfing the web with XProc.

But Andy had run up against a real problem, one best illustrated by an example. Consider the web page shown in Figure 1 [92].

Figure 1. A table of cities



[Home]

# Some cities in the UK

| City | Country | Latitude | Lo |
|---|---|---|---|
| Abbots Langley | England | 51.701 | |
| Aberaman | Wales | 51.7 | |
| Aberbargoed | Wales | 51.6968 | |
| Aberdare | Wales | 51.713 | |
| Aberdeen | Scotland | 57.15 | |
| Abergavenny | Wales | 51.824 | |
| Abergele | Wales | 53.28 | |
| Abertawe | Wales | 51.6167 | |
| Abertillery | Wales | 51.73 | |
| Aberystwyth | Wales | 52.414 | |
| Abingdon | England | 51.6717 | |
| Abram | England | 53.508 | |
| Accrington | England | 53.7534 | |
| Acomb | England | 53.955 | |
| Acton | England | 51.5135 | |
| Addlestone | England | 51.3695 | |

This looks like a perfectly straightforward HTML page. But if we point our document pipeline at it, we get something (that might be) quite unexpected:

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
   <head>
      <meta http-equiv="Content-Type" content="text/html;                    ↩
charset=UTF-8">
      <title>Some cities in the UK</title>
      <script defer src="cities.js"></script>
      <link href="../style.css" rel="stylesheet">
      <link href="cities.css" rel="stylesheet">
      </head>
   <body>
      <p>[<a href="/">Home</a>]</p>
      <h1>Some cities in the UK</h1>

      <table></table>

      <p>Load <button id="more">More</button></p>
   </body>
</html>
```

Right where we expected the table, we get a `table`.

A completely empty table.

The culprit here is the `script` tag on line 7 that loads `cities.js`. When the `p:load` step accesses pages on the web, it does so in much the same way that curl does: it opens a connection to the page, pulls down the data sent by the web server, and hands that back to the processor.

Your browser does the same thing, but it also does a lot more. It downloads all of the linked resources: images, stylesheets, scripts, etc. Then it constructs a styled presentation of the page that includes the images. If scripts were downloaded, those are executed and the page is updated accordingly. Scripting allows a page to be interactive: clicking, selecting, scrolling, the browser supports a huge range of events all of which can cause more script execution and more updates to the page.

All of this is out of reach from the `p:load` step. It wouldn't be hard to extend our document pipeline to find and download linked images, stylesheets, and scripts, but we couldn't *execute* those scripts. We don't have a browser sandbox to run them in.

## 2. Enter Selenium

Selenium's tagline is "Selenium automates browsers. That's it!" Automation works because modern web browsers implement a "WebDriver" set of APIs for this purpose. The main use case is automated testing of web applications, but it's not limited to that.

Selenium is roughly three APIs in a trench coat. One of those APIs talks to the web browser through the WebDriver API. The other exposes a standard Selenium API to a host language. And in the middle, Selenium wires these two APIs together.

There are host language APIs for Java, Python, C#, Ruby, JavaScript, and Kotlin. For an implementation written in any of those languages, talking to Selenium is just a matter of loading the right language bindings.

In theory, we could start a web driver for our browser, start Selenium, direct Selenium to load the page, wait for the scripts running in the browser to populate the table, and ask Selenium to give us the data.

All we need to do is implement a Selenium step.

The trouble with a Selenium step is that it doesn't *do* anything all by itself. It exposes an API that you can drive from a host language. You'd need to be able to drive it from XProc.

To drive Selenium from XProc, the step would need to have some way to describe what the author wanted to do. XProc isn't really designed to support stateful, imperative programming so mapping directly to the Selenium host language APIs isn't really an option.

What's needed is some way for the pipeline author to describe how they want Selenium to behave. They need some mechanism for scripting the interaction. Then we can imagine a Selenium step that takes that script as input and uses it to interact with the browser.

## 3. Enter Invisible XML

The most practical thing seemed to be a little text-based scripting language. Armed with a small amount of experience using Selenium to test web applications and a Selenium reference guide, the plan was:

1. Invent a little bit of syntax.
2. Write a parser for it.
3. See if it works.
4. Repeat until done.

Invisible XML fits perfectly here:

1. Writing parsers in iXML is easy.
2. My processor already supports Invisible XML, so parsing the users script with iXML is easy.
3. The output of a parse is an XML document.
4. In an XProc implementation, writing an interpreter for some XML is easy.

Here's how it started:

```
script version 0.2 .
page "http://example.com" .
```

We start with a version, so we can adapt as it evolves, and then we know we're going to want to load a page. Next, we write a little bit of iXML to parse it:

```
ixml version "1.1-nineml" .

script = versionDecl, s, page, s .

-versionDecl = s, -"script version ", version, s, -"." .
@version = "0.2" .

@page = -"page", RS, string, s, -"." .
```

(I'm eliding a few more lines of grammar needed to parse strings and whitespace. I probably didn't write them for this purpose anyway, I probably copied them from some other grammar.)

Now we just add some more syntax, then some more iXML, and "repeat until done."

It finished as 130ish lines [https://codeberg.org/xmlcalabash/xmlcalabash3/src/branch/main/xmlcalabash/src/main/resources/com/xmlcalabash/ext/selenium-grammar.ixml] of iXML. It took a few evenings, but the resulting language [https://docs.xmlcalabash.com/reference/current/cx-selenium.html#selenium-scripting] supports a healthy subset of the Selenium API, three kinds of conditional blocks, subroutines, variables, and XPath expressions.

(It is in many respects a very sloppy language with side effects and minimal static checking. But it's a proof-of-concept as much as anything.)

## 4. cx:selenium

The `cx:selenium` step [https://docs.xmlcalabash.com/reference/current/cx-selenium.html] is an atomic step:

```
<p:declare-step type="cx:selenium">
  <p:input port="source" content-types="text xml"/>
  <p:output port="result" sequence="true"/>
  <p:option name="browser" as="xs:string?"/>
  <p:option name="capabilities" as="map(xs:QName, item())?"/>
  <p:option name="arguments" as="xs:string*"/>
</p:declare-step>
```

Here's a pipeline that uses `cx:selenium` instead of `p:load` to get a web page. Like our earlier pipeline, this one loads a page and serializes it. But this time, it loads the page with Selenium, allowing the browser to evaluate any scripts it contains:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:cx="http://xmlcalabash.com/ns/extensions"
                name="main" version="3.1">
  <p:import href="https://xmlcalabash.com/ext/library/selenium.xpl"/>
  <p:output port="result"/>
  <p:option name="uri"/>

  <cx:selenium>
    <p:with-option name="arguments" select="('--headless')"/>
    <p:with-input>
      <p:inline content-type="text/plain">script version 0.2 .
      page "{$uri}" .
      pause PT0.5S .
      output to result .
      </p:inline>
    </p:with-input>
  </cx:selenium>
</p:declare-step>
```

Running that script gives us table data:

```
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta charset="utf-8"/>
      <title>Some cities in the UK</title>
      <script defer="defer" src="cities.js"/>
      <link href="../style.css" rel="stylesheet"/>
      <link href="cities.css" rel="stylesheet"/>
   </head>
   <body>
      <p>[<a href="/">Home</a>]</p>
      <h1>Some cities in the UK</h1>
      <table>
         <thead>
            <tr>
               <th>City</th>
               <th>Country</th>
               <th>Latitude</th>
               <th>Longitude</th>
            </tr>
         </thead>
         <tbody>
            <tr>
               <td>Abbots Langley</td>
               <td>England</td>
               <td>51.701 </td>
               <td>-0.416 </td>
            </tr>
            <tr>
               <td>Aberaman</td>
               <td>Wales</td>
               <td>51.7   </td>
               <td>-3.4333</td>
            </tr>
…
            <tr>
               <td>Addlestone</td>
               <td>England</td>
               <td>51.3695</td>
               <td>-0.4901</td>
            </tr>
         </tbody>
      </table>
      <p>Load <button id="more">More</button>
      </p>
   </body>
</html>
```

How does it work? The `page` command loads the page, the `pause` command waits for half a second so the browser has a chance to fill the table, and the `output` command sends the current page DOM to the `result` port.

We can also interact with the page. This script:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:cx="http://xmlcalabash.com/ns/extensions"
                name="main" version="3.1">
  <p:import href="https://xmlcalabash.com/ext/library/selenium.xpl"/>
  <p:output port="result"/>
  <p:option name="uri"/>

  <cx:selenium>
    <p:with-option name="arguments" select="('--headless')"/>
    <p:with-input>
      <p:inline content-type="text/plain">script version 0.2 .
      page "{$uri}" .

      find $button by id = "more" .
      click $button .
      pause PT0.5S .

      output to result .
      </p:inline>
    </p:with-input>
  </cx:selenium>
</p:declare-step>
```

Finds the button on the page with the id “more”, clicks it, waits, then returns the page. That will return the second page of results.

As a final example, consider this script:

```
script version 0.2 .
page "{$uri}" .

until "not(empty($row))" do
  find $row by selector = "table tbody tr" .
  pause PT0.25S .
done

# Search for $city, hit more until we find it
find $city by xpath = "//td[. = '{$city}']".
while "empty($city)" do
  call clickNext .
  find $city by xpath = "//td[. = '{$city}']".
done

find $row by xpath "//tr[td[. = '{$city}']]" .

output xpath "normalize-space(replace($row/*:td[3], ' ', ' '))" to     ↩
result .
output xpath "normalize-space(replace($row/h:td[4], ' ', ' '))" to     ↩
result .

close .

subroutine clickNext
   find $button by selector = "button" .
   scroll to $button .
   click $button .
   pause PT0.25S .
end
```

This script:

1. Loads the page.
2. Waits until the table has data, rather than assuming 0.5s will be long enough.
3. Looks for the city with an XPath expression. While it isn’t present, it uses a subroutine to click the next button.

4. If we've found the city, we get its row.
5. Then we output the latitude and longitude after doing a little cleanup.
6. Then we close the browser.
7. The "clickNext" subroutine scrolls finds the button, scrolls to it, clicks it, and wait's ¼s.

## 5. Security implications

Selenium drives a headless browser. It can essentially do anything with the browser that you can. And it's quite possibly driving a browser that you've already used to login to sites. A malicious script could potentially wreak havoc.

To help mitigate this problem, the `cx:selenium` step can be configured with a whitelist:

```
<x:selenium xmlns:x="https://xmlcalabash.com/ext/ns/selenium"
            whitelist="http://localhost.*
                       https://testdata.xmlcalabash.com/.*"/>
```

With this configuration, the step will only accept URIs that match `localhost` or `testdata.xmlcalabash.com`.

## 6. Next steps

The scripting language could certainly be improved. Someone with deeper Selenium experience might be able to see immediately what's missing. Any attempts to do complex browser interactions are bound to turn up areas where things are awkward at best.

But even as it stands today, it offers new opportunties for surfing the web with XProc.

Markup UK

# PrintCSS Meets LaTeX

## Implementation of a PrintCSS Renderer Prototype Using a CSS Parser, an XML-to-TeX Conversion Framework, and a LuaTeX-Based Framework

**Martin Kraetke, le-tex publishing services GmbH**

**Christine Windeln, le-tex publishing services GmbH**

## 1. Introduction

At Markup UK 2023, I discussed various methods for converting XML to TeX and presented our own approach using the transpect library xml2tex, which can be configured to convert arbitrary XML into TeX [1]. xml2tex and its associated libraries are based on XProc and XSLT and are used at le-tex for several applications, such as a converter for converting Word to TeX, an equation renderer, and for our typesetting system *xerif*[2].

For the latter, the schema-independent configurability of xml2tex helps to meet the diverse content requirements of our *xerif* clients. The configuration allows us to use several different XML schemas and map them to customer-specific TeX macros. While the xml2tex configuration allows us to map XML to TeX very flexibly with *xerif* , the layout setup, e.g. setting up the page format, selecting fonts, specifying spacing, etc. is still done conventionally in TeX.

In *xerif*, CoCoTeX serves this purpose. It’s a custom-tailored TeX framework that facilitates the setup of these parameters and extends LaTeX with numerous custom macros, a custom metadata and table model, accessibility support, and more. However, TeX’s complexity and the steep learning curve of our CoCoTeX framework presenting a major hurdle for external newcomers attempting to use *xerif* effortlessly.

A language for formatting page-based media that is considered much easier to learn is PrintCSS. Unlike TeX, PrintCSS is accessible and user-friendly, leveraging the familiar CSS syntax. This approach has led to its widespread adoption in automated publishing workflows. PrintCSS introduces a set of CSS specifications tailored for print media, enabling precise control over page layouts, content fragmentation (such as pages, columns, or regions), and content generation (including running headers, page numbers, and listings). By utilizing PrintCSS, HTML or XML content can be formatted for paged media output in a manner akin to styling conventional web pages with CSS.

However, it's important to note that PrintCSS has certain limitations: the existing specifications lack specific features, and in some areas, they remain underspecified. Moreover, there are only a limited number of implementations of these specifications, and the two available open-source renderers implement only a subset of the common CSS paged media features. Instead, these implementations address the absence of certain functionalities through the use of proprietary extensions. Leading developers of browser

[1]1 Martin Kraetke (2023): Bridging the Gaps Between XML and TEX. Available at https://markupuk.org/2023/webhelp/index.html (Accessed: May 15, 2025)
[2]le-tex publishing services (2025): xerif. automatic is better. Available at: https://www.le-tex.de/en/xerif.html (Accessed: May 15, 2025)

engines—namely Google, Mozilla, and Apple—do not provide support for PrintCSS in their browsers and show little motivation to include this feature in the future.

Nevertheless, the advantages of PrintCSS's ease of learning outweigh the disadvantages of its ambiguous specifications, so we sought to investigate the feasibility of incorporating PrintCSS support within the *xerif* framework. Starting from October 2024, Christine took on the task as part of her bachelor thesis and was able to implement a prototypic support for PrintCSS in *xerif* . This paper aims to outline the steps undertaken to implement their solution, discuss how we addressed the conceptual and technological differences between PrintCSS and TeX, and identify potential directions for future research.

## 2. The Status Quo: Configuring Layouts in *xerif*

To understand how the layout configuration works, we have to take a look at the TeX preamble of a particular TeX document created with *xerif* . The layout features and default settings are provided by the `cocotex` document class. The customer-specific configuration is represented by a custom TeX style that is usually named after the client, e.g. `brill` for Koninklijke Brill NV, or now De Gruyter Brill. Macros for tables are provided by `h tmltabs` , a package for specifying tables in a HTML-like fashion.

```
\documentclass[greek,main=english,pubtype=collection]{cocotex}
\usepackage{htmltabs}
\usepackage[layout=1]{brill}
```

The CoCoTeX framework is separated into various modules that provide specific macros for various layout aspects:

- Kernel: default macros to declare and evaluate CoCoTeX macros and properties
- Common: default settings and required TeX packages
- Floats: images and tables
- Frame: page boxes, e.g. trim box, bleed box and crop marks
- Accessibility: support for PDF/UA
- Headings: part, chapter and section headings and their metadata
- Lists: provides macros for various list types
- Meta: macros for document metadata
- Notes: endnotes and footnotes
- Script: default font settings
- Title: Macros and settings for title pages

The custom TeX style is built upon the features provided by CoCoTeX and sets customer-specific parameters such as margins, fonts, type area, font sizes, leading and other styles. It’s also possible to define customer-specific macros or overwrite existing CoCoTeX macros. The style files can be selectively overridden by a cascading configuration. For example, you could specify different TeX styles based on the imprint, book series or individual books if necessary but our developers use this approach rarely.

For TeX developers, using TeX is obviously the most natural way to specify layout parameters. However, a normal TeX user would hardly be able to create a layout with *xerif* .

Unfortunately, it is not enough to be able to write a paper with LaTeX. You need to have advanced TeX programming skills, be familiar with many libraries in the TeX ecosystem and know the CoCoTeX basics. For TeX novices, the learning curve might quickly become as steep as a wall.

## 3. PrintCSS and TeX

Of course, one of the main reasons to use TeX for *xerif* was that TeX is one of the technological building blocks of le-tex. The company also has little experience with other typesetting technologies such as XSL-FO and PrintCSS. But aside from the general tendency towards self-justification bias, there are valid reasons to prefer TeX over PrintCSS:

In contrast to TeX, CSS is not a programming but a formatting language. There is less you can do with PrintCSS and you are dependent on the functionality being provided by a layout engine, the PrintCSS renderer. PrintCSS has a quite smaller feature set compared to TeX and its vast ecosystem of packages. Although PrintCSS, unlike TeX, has a public specification, many things are missing that must be substituted by proprietary functions of the PrintCSS renderer.

PrintCSS might not be as intuitive as InDesign, but its core syntax and principles are beginner-friendly especially if you gained prior knowledge of CSS by web design. It provides a solid foundation for defining page-based layouts, and limitations in the specification could be addressed through vendor-specific CSS extensions.

TeX is a powerful typesetting engine capable of producing high-quality print documents and it might be tempting to think of TeX acting exclusively as a PrintCSS renderer. It is worth recalling that PassiveTeX, created by Michel Goossens and Sebastian Rahtz, served as an experimental solution for leveraging TeX as an XSL-FO formatter [3]. But is TeX also capable of being used as PrintCSS renderer?

Conceptually, mapping CSS properties to TeX equivalents (e.g., margins, page breaks, fonts) is possible. However, this approach present a few practical challenges:

1. Different paradigms: CSS is box-based and declarative; TeX is macro-based and procedural. A wide range of differences exist also between PrintCSS and TeX, from the underlying page model and font characteristics to the treatment of typographic issues.

2. Limited parsing capabilities: TeX doesn't natively understand XML, HTML or CSS.

3. Missing Features : CSS features like flexbox, grid, or advanced selectors have no direct TeX equivalent.

In this context, the three primary challenges involve parsing PrintCSS, transforming HTML to TeX, and accurately mapping CSS styling rules to corresponding TeX commands. However, relying solely on TeX doesn't seem practical for several reasons:

Creating a custom PrintCSS parser in TeX would be necessary, and the same challenge arises when handling HTML or XML. While TeX-based tools like xmltex could be used for XML parsing, they provide only limited configurability and do not support validation. Because of its procedural nature, TeX is not well-suited for navigating complex tree-based structures like XML and generating mixed content. So, there are additional layers needed, to facilitate the transformation between PrintCSS and TeX. The next section provides more details.

[3]2 Martin Kraetke (2023): Bridging the Gaps Between XML and TEX. PassiveTeX. Available at https://markupuk.org/2023/webhelp/index.html#Sec2.html (Accessed: May 15, 2025)

# 4. Building a PrintCSS renderer

## 4.1. Preliminary Considerations

First it was necessary to define the scope – on the one hand, to keep the work within the limits of a bachelor's thesis, and on the other hand, to provide a solid feature set that make it feasible for practical use. The most important specifications for PrintCSS have all been taken into account:

- CSS Paged Media Module Level 3 [4]
- CSS Fragmentation Module Level 3 [5]: partitions a flow into pages, columns, or regions,
- CSS Generated Content for Paged Media Module [6]: running headers, footers and footnotes
- CSS Page Floats [7] specifies page floats that can be shift to the top or bottom

For a detailed list of the included CSS features, please refer to page Section 6 [107]. However, there are various other CSS specifications involved, which are necessary for specifying fonts, applying colors and text styles, define font sizes and line heights etc. We defined a standard feature set which is listed at page Section 7 [107].

## 4.2. Parsing PrintCSS

Christine took on the majority of the work by independently implementing and testing the code, while occasionally consulting with Martin throughout the process. But she did not need to start from scratch. A CSS parser was already part of our transpect framework, but had to be extended for the PrintCSS grammar.

The CSS parser is based on an EBNF schema. EBNF is the abbreviation of Extended Backus-Naur form and is a declarative syntax to express a syntax (metasyntax) of a formal language. Christine has extended our CSS EBNF schema. For example, here is an EBNF code snippet that declares the syntax for specifying page rules in CSS:

```
pagerule ::= '@page' S* (pageclass | pagename)?
pagename ::= IDENT
pageclass ::= ':' ('first'|'blank'|'left'|'right')
```

Using Gunther Rademacher's REx Parser Generator [8], the EBNF was later converted to XSLT. The XSLT is applied on the CSS file and creates an XML document including the CSS rules. The example below shows the XML representation (CSS XML) of a page rule:

```
<atrule origin="/tmp/style.css" type="page">
  <raw-css xml:space="preserve">
    @page {&#xD;
      size:A4;&#xD;
      margin: 2cm 2cm 3cm 2cm;&#xD;
```

[4]3 Elika J. Etemad et. al (2023): CSS Paged Media Module Level 3. W3C Working Draft. Available at https://www.w3.org/TR/css-page-3/ (Accessed May 15, 2025)
[5]4 Rossen Atanassov and Elika J. Etemad (2018): CSS Fragmentation Module Level 3. W3C Candidate Recommendation. Available at https://www.w3.org/TR/css-break-3/ (Accessed May 15, 2025)
[6]5 Mike Bremford and Rachel Andrew (2024): CSS Generated Content for Paged Media Module. W3C Working Draft. https://www.w3.org/TR/css-gcpm-3/ (Accessed May 15, 2025)
[7]6 Johannes Wilm (2015): CSS Page Floats. W3C First Public Working Draft. Available at https://www.w3.org/TR/css-page-floats-3/ (Accessed May 15, 2025)
[8]7 Guter Rademacher (2025): REx Parser Generator. Available at https://github.com/GuntherRademacher/rex-parser-generator (Accessed May 15, 2025)

```
    }
  </raw-css>
  <declaration property="size" value="A4"/>
  <shorthand property="margin" value="2cm 2cm 3cm 2cm" num="2"/>
  <declaration property="margin-top" value="2cm" shorthand="2"/>
  <declaration property="margin-right" value="2cm" shorthand="2"/>
  <declaration property="margin-bottom" value="3cm" shorthand="2"/>
  <declaration property="margin-left" value="2cm" shorthand="2"/>
</atrule>
```

Later, the CSS rules are applied to the content by evaluating the CSS selectors and inserted back into the source file as CSS attributes (CSSa) [9]:

```
<p>The result of a Delta-DOR measurement provides knowledge
of the spacecraft’s angular position in the inertial reference frame ↩
defined
by the quasars (CCSDS <a href="#j_astro-2022-0200_ref_002" >2019</ ↩
a>).</p>
```

This CSS parser is encapsulated in a simple XProc step, which takes an XHTML document and an XSLT parser stylesheet as inputs. It produces both the CSS XML representation and an XHTML document with embedded CSS attributes (XHTML/CSSa):

```
<css:expand>
  <p:input port="source" primary="true"/>
  <p:input port="stylesheet"/>
  <p:output port="result" primary="true"/>
  <p:output port="xml-representation"/>
  <p:output port="report" sequence="true"/>
</css:expand>
```

### 4.3. Generating an xml2tex Configuration

How is this information converted to TeX? Our xml2tex library is for us a natural choice for several reasons: it provides a schema-independent grammar for transforming XML into TeX, and it includes built-in libraries for handling formulas and tables, eliminating the need for custom solutions. Moreover, it seamlessly integrates with our *xerif* typesetting system.

As in CSS, in TeX some formatting information is inserted inline into the text, while others are typically declared in an external style file. For example, there are general formatting rules that specify the layout of the page and other rules that set a portion of text in italics. For this reason, Christine needed to analyze these layers and separate them systematically. Thankfully, xml2tex is able to declare different outputs with one configuration, e.g. a TeX file and a style file.

So, we decided to generate a xml2tex configuration with XSLT by analyzing XHTML/CSSa and CSS XML. The first part of the generated configuration is the creation of the TeX style file. Paper format, font declarations, font styles, headline elements (h1…h6), etc. are evaluated and the respective TeX macros are created. To align with *xerif* , the TeX style includes many instructions from our CoCoTeX framework. For example, by analyzing the size property of the `@page` rule above, this TeX snippet is created:

[9]8 Gerrit Imsieke (2013). Conveying Layout Information with CSSa (Talk held at XML Prague). Available at https://archive.xmlprague.cz/2013/presentations/Conveying_Layout_Information_with_CSSa/CSSa_xmlprague_gimsieke.html#/step-1 (Accessed May 15, 2025)

```
<template context="/html:html/html:head/(html:style|html:link)[1]">
  <file href="file:///home/cwindeln/printcss2tex/tmp/printcss.sty"    ↩
encoding="utf-8">
% (...)
\setlength{\pagewidth}{210mm}
\setlength{\pageheight}{297mm}
% (...)
  </file>
</template>
```

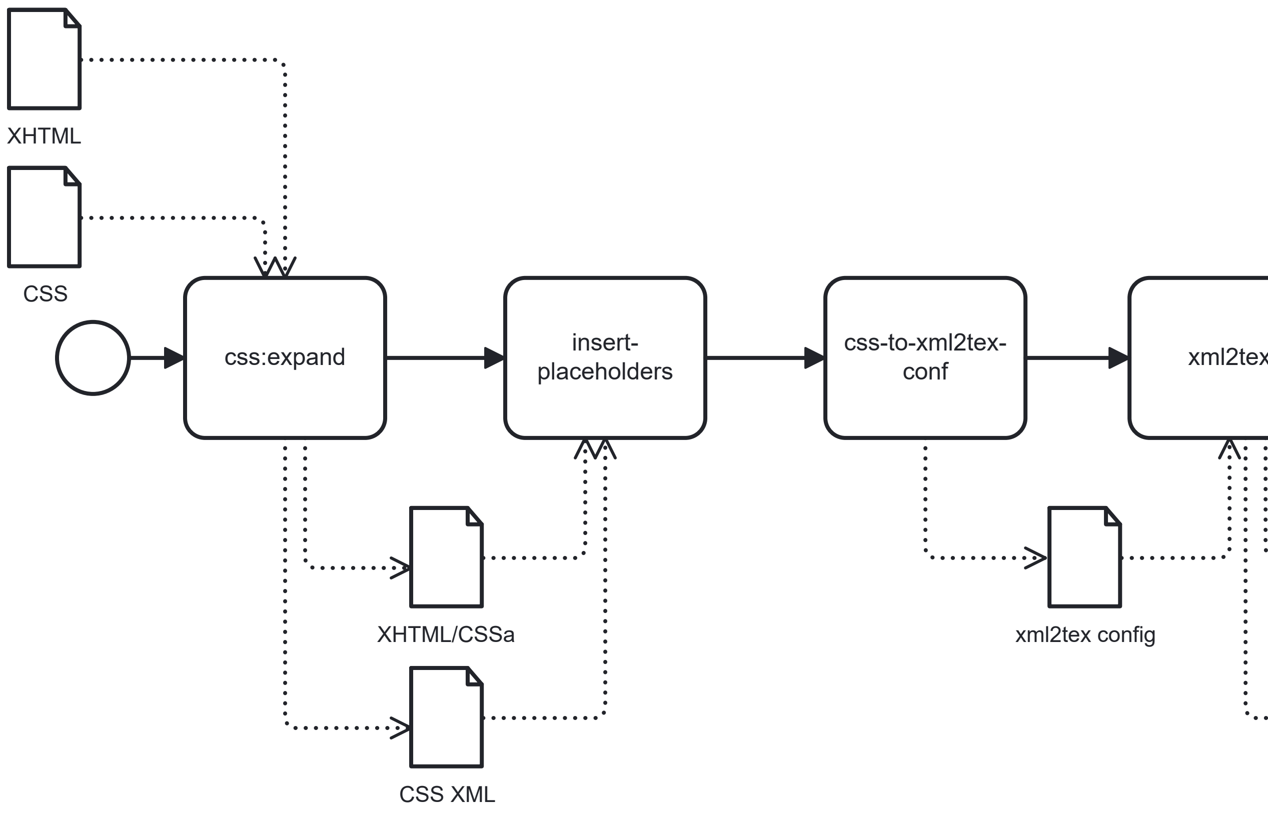
The second part of the configuration is generated by analyzing the CSS declarations and creating xml2tex templates from it. The xml2tex template below is generated from a declaration which sets the font size for `<p>` elements:

```
<template context="*:p">
  <text>{\fontsize{11.5pt}</text>
  <xsl:next-match xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
  <text>}</text>
</template>
```

During the implementation, Christine noticed that there is no equivalent for the CSS selectors `::before` and `::after` in TeX. Therefore, we decided to create placeholder elements before, into which we can later generate the content using xml2tex. The entire transformation is organized into distinct steps, as illustrated below:

1. Parse the CSS and generate CSS XML and XHTML/CSSa
2. Insert pseudo elements for content that is inserted with the CSS selectors `::before` and `::after`
3. Generate a xml2tex configuration from CSS XML and XHTML/CSSa
4. Use xml2tex with the generated configuration and apply it on the XHTML/CSSa document to create TeX and the style file.
5. Render the TeX file with LuaTeX

Figure 1. Figure 1: The transformation of XHTML and CSS to TeX



## 5. Summary, Discussion and Future Work

Christine was able to write a pipeline that can create a PDF from HTML and CSS by extending our CSS parser and using libraries that are used in *xerif* such as xml2tex and CoCoTeX. The presented software tool provides fundamental support for the PrintCSS specifications and could provide a basis for our clients to easily create their own layouts using PrintCSS.

However, due to the inherent differences between CSS and TeX, certain limitations in implementation may arise. In particular, complex layout features or specific typographic refinements may not be fully realized according to the PrintCSS standards. For example, while CSS allows you to specify the maximum number of orphans in a paragraph, in TeX you would specify a penalty. Multicolumn layouts work slightly differently in CSS and TeX and we did not introduce advanced CSS layout models like grid and flexbox.

Furthermore, CoCoTeX offers many layout features that are not possible with standard PrintCSS todays like a class system for figures and tables, improved word- and line-breaking options, custom dictionaries, advanced listings (ToC, LoT, LoF) and a robust PDF/UA-compliant accessibility support. Many of the CoCoTeX features did not make it yet into our PrintCSS renderer, but we plan to implement them as vendor-specific extensions in the future. More advanced CSS features need to be evaluated and might be implemented in TeX.

In any case, our goal isn't to cover the entire CSS specification. As with other PrintCSS formatters, there will always be limitations in the supported CSS vocabulary, which won't be particularly significant for users as long as they can realize their typographic ideas. After all, there is no web browser that support PrintCSS, and we certainly don't need CSS animations for *xerif*. Looking ahead, we aim to further explore the boundaries of what TeX can represent in the context of PrintCSS.

## 6. Appendix I: Supported PrintCSS Features

| Item | Limitations | Spec |
|---|---|---|
| `@page` rule | | CSS Paged Media Module Level 3 |
| page selectors | `:blank` | |
| `@(top\|right\|bottom\|left)- ((left\|right)-corner)\| (top\|right\|middle\|center\|left\|bottom)` | | |
| `size` | | |
| `page-orientation` | | |
| `marks` | | |
| `bleed` | | |
| `break-(after\|before\|inside)` | | CSS Fragmentation Module Level 3 |
| `page-break-(after\|before\|inside)` | | |
| `orphans` | | |
| `widows` | | |
| `string-set` | | CSS Generated Content for Paged Media Module |
| `@footnote` | | |
| `footnote-call` | | |
| `footnote-marker` | | |
| `float` | `only block-start\|top\|bottom` | CSS Page Floats [a] |
| `float-reference` | `only column\|page` | |

[a]9 Johannes Wilm et. al (2024): CSS Page Floats. W3C Editor’s Draft. Available at https://drafts.csswg.org/css-page-floats/ (Accessed May 15, 2025)

## 7. Appendix II: Supported Standard CSS Features

| Item | Limitations | Spec |
|---|---|---|
| selectors | only Universal, `#id`, `.class`, `::before`, `::after`, `::first-line`, `::first-letter` | CSS Selectors Level 3 [a] |
| `@font-face` rule | only OTF fonts supported | CSS Fonts Module Level 3 [b] |
| `font-family` | | |
| `font-size` | | |
| `font-style` | | |
| `font-variant-position` | | |
| `font-variant-caps` | | |
| `font-weight` | | |
| `margin(-(top\|right\|bottom\|left))?` | | CSS Box Model Module Level 3 [c] |
| `padding(-(top\|right\|bottom\|left))?` | | |
| `width`, `height` | | CSS Box Sizing Module Level 3 [d] |
| `min-width`, `min-height` | | |

| | | |
|---|---|---|
| `border(-(top\|right\|bottom\|left))?(-(color\|style\|width))?` | | CSS Backgrounds and Borders Module Level 3 [e] |
| `background(-color)?` | | |
| `color` | | CSS Color Module Level 3 [f] |
| `text-align` | | CSS Text Module Level 3 [g] |
| `text-indent` | | |
| `line-break` | | |
| `hyphens` | | |
| `white-space` | | |
| `counter-reset` | | CSS Lists and Counters Module Level 3 [h] |
| `counter-increment` | | |
| `list-style(-type)?` | | |
| `bookmark-(level\|label\|state)` | | CSS Generated Content Module Level 3 [i] |
| `leader()` | | |
| `target-counter()` | | |
| `target-text()` | | |

[a]10 Tantek Çelik et. al (2018): Selectors Level 3. W3C Recommendation. Available at https://www.w3.org/TR/selectors-3/ (Accessed May 15, 2025)

[b]11 John Daggett et. al (2018): CSS Fonts Module Level 3. Available at https://www.w3.org/TR/css-fonts-3/ (Accessed May 15, 2025)

[c]12 Elika J. Etemad (2024): CSS Box Model Module Level 3. W3C Recommendation. Available at https://www.w3.org/TR/css-box-3/ (Accessed May 15, 2025)

[d]13 Tab Atkins and Elika J. Etemad (2024): CSS Box Sizing Module Level 3. W3C Working Draft. Available at https://www.w3.org/TR/css-sizing-3/ (Accessed May 15, 2025)

[e]14 Elika J. Etemad and Brad Kemper (2024): CSS Backgrounds and Borders Module Level 3. W3C Candidate Recommendation Draft. Available at https://www.w3.org/TR/css-backgrounds-3/ (Accessed May 15, 2025)

[f]15 Tantek Çelik et. al (2018): CSS Color Module Level 3. W3C Recommendation. Available at https://www.w3.org/TR/css-color-3/ (Accessed May 15, 2025)

[g]16 Elika J. Etemad et. al (2024): CSS Text Module Level 3. W3C Candidate Recommendation Draft. Available at https://www.w3.org/TR/css-text-3/ Accessed May 15, 2025)

[h]17 Elika J. Etemad and Tab Atkins (2020): CSS Lists and Counters Module Level 3. W3C Working Draft. Available at https://www.w3.org/TR/css-lists-3/ (Accessed May 15, 2025)

[i]18 Elika J. Etemad and David Cramer (2019): W3C Working Draft. Available at https://www.w3.org/TR/css-content-3/ (Accessed May 15, 2025)

# Adding new Cars to a Running Train

## Adding markup for multilingual documents to JATS

Deborah Aleyne Lapeyre, Mulberry Technologies, Inc.

B. Tommie Usdin, Mulberry Technologies, Inc.

The Journal Article Tag Suite (JATS) is the ANSI/NISO standard that is defines the tag set used to archive, exchange, and publish journal articles worldwide. JATS 1.4 was approved (as JATS version 1.4 ANSI/NISO Z39.96-2024) on October 31, 2024. The most exciting new capability added in JATS 1.4 is a way to encode multi-language journal articles. JATS defines a “multi-language” article not merely as an article written in English or German with a couple of quotations in Latin or French. A true multilingual article contains substantial portions of the content written and presented in more than one language or contains some parts in one language and other parts in a different language or languages. The JATS tag set has always been able to encode articles in any language and to identify portions of an article that are in a different language from the containing article (using @xml:lang). However, JATS has not, until version 1.4, been able to encode (in a graceful way) articles that are in whole or large-part multilingual.

The JATS multi-language mechanism harks back to Architectural Forms, and is implemented mostly using attributes, which can identify:

- an entire article in 2 or more languages
- substantial portions of content (paragraphs and sections) replicated in 2 or more languages
- block structures (such as figures, tables, and equations) replicated in 2 or more languages
- articles where some of the content is in one language and some in another, and
- article metadata (such as the article title or abstract) in multiple languages.

This paper describes the encoding available for many styles of multi-language functionality, provides examples, lists useful resources to learn more, and gives a few reasons why this mechanism may be of interest to you, even if you or your clients do not code in JATS or work with journal articles.

## 1. What is JATS?

The Journal Article Tag Suite (hereafter JATS) is an ANSI/NISO standard (JATS version 1.4 ANSI/NISO Z39.96-2025) designed to capture the content and metadata of journal articles. JATS is only a tag set, unlike TEI, DITA, and Akoma Ntoso, which provide extensive software/processing components. JATS provides semantic tags for the parts of journals (particularly in the metadata) that have traditionally been used for discovery and to manage journal articles. The JATS tagging for body structures is quite generic (paragraphs, tables, figures, lists, footnotes, etc.). The metadata and bibliographic elements are typically semantic and divide many elements into logical components (for example: multiple parts of a personal name, extensive information on collaborations, elaborate article funding metadata, and over 60 elements within a bibliographic citation).

Journal articles are *not* typically authored in JATS: they may be, for example, written in Microsoft Word or encoded in a bespoke journal article tag set. But at some stage in their lifecycle nearly all the world's journal articles are translated into JATS. JATS is the tag set publishers and archives use to communicate and exchange; JATS can be input to a wide variety of hosting platforms; JATS is the archival format of choice for many large-scale archives such as PubMed Central, UK PubMed Central, JSTOR, the British National Library, the Australian National Library, and the US Library of Congress. JATS is international; in 2018 it was used in more than 30 countries, and that number grows over time.

## 2. Requirements and Constraints

Historically, publishing in general, and journal publishing in particular, has been dominated by a relatively small portion of the world and by European languages and English in particular. This is changing; there are not only increasing numbers of journals in other languages, there is a growing need to support multilingual journal articles. Multilingual content is much more than block quotes in French or Latin inside a article in English. Familiar examples from current publishing practice include an article in two original languages (the Canadian author wrote both French and English versions) or in an original language and one or more translations (a French/English article was also translated into Spanish).

### 2.1. How Multi-language Documents are Presented and Stored

Both the presentation styles and the order of storing components are widely variable for current multi-language documents. When substantial portions of document content are in more than one language, the same structures are often repeated, once in each language. For example: alternate sections could repeat the same content in Spanish and Portuguese. As another, a paragraph could be repeated, first in Greek, then in Romanian, and then in Italian. Sometimes an article will be almost entirely in one language with selected structures in both (or only) another language, as article written in Korean with both metadata and table content provided in English. [[JATSMultilang]]

In such a multilingual document, when there are two or more “same-content” structures (sections, figures, boxed-texts, tables, etc.), differing only in language, few assumptions can be made about the locations of and the interrelationships between these same-content objects within the document.

- content objects written in a single language need not be contiguous, and
- equivalent structures (the same content, differing only by language) need not be located anywhere near each other in the document

For these reasons, a JATS multi-language mechanism could not simply enclose (wrap up) all the same-content objects, unless the entire article was presented, in order, twice. This wrapper-approach has been the mechanism of the long-standing JATS element `<block-alternatives>`, which was created to hold multiple copies of a block object such as a figure or table in multiple languages. Users tried to use `<block-alternatives>` for multilingual documents and found that it did not meet real-life needs. For true interspersed multilingualism, a wrapper-style mechanism is not sufficient. [[JATSMultilang]]

## 2.2. Requirements and Non-Requirements

The scope and culture of JATS impose some requirements and constraints on the ways in which JATS can or should grow. The most important of these are that within the 1.X line strict backwards compatibility is essential, inconvenience to current users should be minimized, and JATS is a source format for storage and interchange rather than a display format.

### 2.2.1. Backwards Compatibility

The current version of JATS is 1.4. Like all versions in the 1.X line, it is fully backwards compatible with all previous versions of JATS. This means that any document valid to one of the JATS profiles (Archiving, Publishing, Authoring) in any of the previous versions of JATS will be valid to the current version. The JATS Committee has further committed to the position that all previous valid JATS 1.X documents will be valid to future 1.x versions of that profile of JATS.

While there has been discussion of what a 2.0 version of JATS might look like, and what infelicities in JATS might be fixed if /when we can tidy up by removing structures that are no longer optimal, JATS 2.0 is in the indefinite future. So, accommodation of multilingual articles had to be built into JATS 1.X, and thus in a way that articles that were created before this was developed would be valid and would have no changes to their meanings when used with a schema that included the multi-language mechanism.

### 2.2.2. The Needs of the Few Must Not Burden the Many

Most journal publishers do not publish multilingual articles, most journal articles are not multilingual, and most JATS users have not been uncomfortable because JATS did not accommodate multilingual articles. It was important that, in adding a mechanism to enable a small but growing community to use JATS, we not impose a burden on established users. (The JATS Standing Committee was told, in no uncertain terms, that “NOBODY NEEDS THAT #*($!”. We interpreted that to mean that the speaker didn't need it, wasn't interested in people who might need it, and was concerned about the burden that might be imposed on existing users if it were added.)

It was the hope of the JATS Standing Committee that the JATS multi-language mechanism be neither bulky nor intrusive. Ideally, any mechanism should enable users to create true multilingual documents while not requiring changes to the tagging of mono-lingual documents. Ideally, any multi-language mechanisms should be completely ignorable by creators/users of mono-lingual documents. In other words, if you do not publish multilingual documents, nothing in your JATS-world should change.

The JATS 1.4 multi-language structures were designed to be TOTALLY IGNORABLE. Users who have no need for multiple languages in one document can simply not use any of them. Some users make their own subsets of the JATS model that do not include *any* of these attributes. They can continue to do so and to leave the multi-language structures out of their working versions. Documents that are valid to such a subset will be valid to the published models and will be indistinguishable from documents that were created using the whole model that happen not to use the Multi-Language structures. [[JATSMLExamples]]

Further, it is highly unlikely that any user will ever use *all* of the multi-language attributes at all, and never within a single article (with the possible exception of articles created to demonstrate use of the mechanism). The multi-language attributes can and should be used only when appropriate and useful, and when the information is available. A user should use the multi-language attributes, like everything else in JATS, to encode information that they desire to provide to users of the articles. There is no virtue, and significant cost, in encoding information in the XML that is unimportant, redundant, or of questionable accuracy. In other words, if:

- you don't know, or
- you don't care and don't think your users will either

don't clutter your XML. The multi-language mechanism is a tool for communicating multilingual information that matters rather than a requirement that creators fill their documents with guesses or information that is irrelevant to the intended use/users of the articles. [[JATSMLExamples]]

#### 2.2.3. JATS Should Stay In It's Lane

**JATS is a source for many presentations, not a display format.**

It is the goal of JATS to enable users to encode the information that will be needed to create a variety of interchange and presentation formats from the markup in the JATS-encoded document. It is not the intent for JATS-encoded articles to be directly ingested by typesetting, voice-synthesis, or web displays. So, for example, JATS provides tagging for both short/unstructured and longer/structured annotations (to a graphic, for example) to enhance accessibility. It does not position this information in the locations that are appropriate for end user display; the assumption is that the application creating the display will put the information where it is needed for display. (For example, some guidelines suggest that HTML's “Long Description” should be on a separate HTML page from the content. That content is embedded in the JATS-encoded article but can be separated when creating a web version of the document.)

Similarly, many page layout tools require heading levels be explicit in their input source; first level heads must have a different tag from second- and third-level heads. JATS uses a containment hierarchy to store this information: the “title” of a “section” that is inside only one other “section” would be displayed as a second-level head. It is the assumption behind the design of JATS that the application providing the JATS-encoded content to that typesetting system will convert the tagging to the style required by that system.

The JATS multi-language tagging includes some of the information described in the W3C's “Internationalization Tag Set (ITS)”. For example, JATS enables users to encode information that can be converted to ITS':

- translate,
- language information, and
- provenance.

JATS does not provide tagging for information that is specific to display tools or that is unlikely to appear in published journal articles, such as:

- localization note,
- localization quality,
- allowed characters, and
- storage size.

## 3. I Don't Use JATS; Why Should I Care?

There are two things a non-JATS user might find of interest in how we addressed this new requirement in JATS:

- a mechanism for adding significant functionality to a existing tag set without burdening existing users, and

- some of the requirements for multilingual documents.

The days when every markup project started with the development of a bespoke vocabulary are far behind us. Shared vocabularies dominate markup applications, often vocabularies that are long lived and used by a wide variety of users. The proponents of many of these vocabularies have evangelical tendencies; they want more and more users to adopt their vocabulary, and in order to make that possible they continually enrich the vocabulary both to better meet the needs of current users and to meet the needs of possible future users. It is important to do this is in ways that do not sabotage the utility of the vocabulary for current users.

Writers in so-called “minority languages”, writers who often find that it is useful to use multiple languages to communicate, are gaining traction. More and more of them are rejecting the premise that in order to publish they must use one of the majority languages as must their readers.

While it seems unlikely that other communities will find that the structures JATS developed to meet this need will be exactly what they need, we hope that others will find our analysis helpful as you figure out how to accommodate these new requirements and new users.

## 4. JATS Multi-language Mechanism (`@lang-group`)

The basic idea is to enable JATS creators to encode structural and metadata components of their document (sections, paragraphs, figures, quotations, tables, etc.) as being the same content, differing only in language. JATS can also encode the relationship between the same content in different languages. (For example: The original author wrote the same content in two or more languages. This section is present twice: one is the original and the second is its translation by a human translator. This section is present twice: one is the original and the second was translated by computer translation.)

In the JATS 1.4, same-content structures in different languages can be flagged as belonging to the same “language group”. JATS defines “language group” as the collection of objects that are the same in content and vary in language. JATS calls alternate language versions of the same content “variants”, and they are collected into a language group by the values of the `@lang-group` attribute. [[JATSMultilang]] The members of a language group may appear in widely separated places within a document.

Functionality dealing with language groups in an online environment might include: allowing the user to choose whether to see only a particular language version or all the variants, and allowing a user to find an article in a search using a filter specifying the language(s). For example, in an article in both English and Spanish, a user could opt to see only the Spanish, only the English, or both. This would be a function of the display application supported by the JATS markup. [[JATSMultilang]]

How an attribute builds language groups: [[JATSMultilang]]

- The value of the `@lang-group` attribute is an `IDREF`, and the attribute uses that value to logically tie the members of a language group (all the variants) together. The variant content objects in the language groups are bound together only by the `IDREF` value.

- The value of the `@lang-group` must be the same for all members of a language group to support processing.

- The value of the `@lang-group` must be the `@id` attribute value of one of the variant objects in the group. (It does not matter which object, and there is no significance to the selection of which `@id` is used.)

Once the objects in the language group are identified the creator can provide a variety of information about each of them. It would be surprising if the language of each were not provided, although it is not required. In addition, information about the source of the content and the expected uses may be provided.

### 4.1. Influence of SGML's “Architectural Forms”: Giving Credit where Credit is Due

This mechanism will look very familiar to old-timers in the markup community. Using a bundle of attributes to attach information to document structures that can govern processing is at the heart of SGML's “architectural forms”. Architectural forms are key to HyTime and DITA, and have been used in many other markup environments.

While the JATS multi-language structures do not use architectural forms as defined in the 1990s, their design was heavily influenced by them. (Most of the publications about SGML architectural forms are no longer available. The best reference we have found that is available is “A Reader's Guide to the HyTime Standard” [[HyTime]].

## 5. A Few Examples

These examples are provided to show the reader a few of the types of information that can be encoded using the JATS multi-language mechanism. See the information sources discussed below in “Documentation for All This” for more detail.

### 5.1. Simple Example of a Language Group

Here is the same product note three times (In English, in French, and in Canadian French): [[JATSMLExamples]]

```
<p id="mug-665" lang-group="mug-665" xml:lang="en">Portable
     battery life varies by product. Enjoy
     all-day battery life by regularly placing mug on its
     included charging coaster.</p>

     <p lang-group="mug-665" xml:lang="fr">L'autonomie de la
     batterie varie en fonction des produits.
     Prolongez l'autonomie de votre mug en le plaçant
     régulièrement sur son support de chargement
     inclus.</p>

     <p lang-group="mug-665" xml:lang="fr-CA">L'autonomie de la
     pile portable varie selon les produits.
     Profitez d'une pile qui dure toute la journée en plaçant
     régulièrement la tasse sur le sous-verre de
     recharge inclus.</p>
```

### 5.2. Did a Person or an Algorithm Translate This?

Three short paragraphs are part of a language group. Note the differences in the treatment of the onomatopoeia (the speech of the hen). The paragraph in English is the original variant: [[JATSMLExamples]]

```
<p xml:lang="en"
     id="LRH-4011" lang-group="LRH-4011" lang-source="author"
     lang-variant="original">Carrying the sack of Wheat, she trudged
     off to the distant mill. There she ordered the Wheat ground into
     beautiful white flour. When the miller brought her the flour
     she walked slowly back all the way to her own barnyard in her
     own <named-content content-type="quote">
     picketty-pecketty</named-content> fashion.</p>
```

The the same paragraph was translated into Welsh by a human translator: [[JATSMLExamples]]

```
<p lang-group="LRH-4011" xml:lang="cy"
     lang-source="translator"  lang-variant="translation">
     Dan gario'r sach Gwenith, trampiodd i ffwrdd i'r felin bell.
     Yna, gofynnodd am falu'r Gwenith yn flawd gwyn, hyfryd. Pan
     ddaeth y melinwr â'r blawd iddi, cerddodd yn araf yn ei hôl
     i'w beudy ei hun, yn ei ffordd <named-content
     content-type="quote">piceti-peceti</named-content>
     ei hun.</p>
```

The the same paragraph was also translated into Welsh by Google Translate: [[JATSMLExamples]]

```
<p lang-group="LRH-4011"  xml:lang="cy"
     lang-source="custom" lang-source-custom="GoogleTranslate"
     lang-variant="translation">
     Gan gario'r sach Gwenith, ymlwybrodd i'r felin bell. Yno hi
     a orchmynnodd y tir Gwenith yn flawd gwyn hardd. Pan y daeth
     melinydd â'r blawd iddi cerddodd yn araf yn ôl yr holl
     ffordd iddi iard ysgubor ei hun yn ei ffasiwn
     <named-content
     content-type="quote">piced-pecedi</named-content>
     ei hun.</p>
```

## 5.3. Not Everything *Should* be Translated

Sometimes some text should not be translated at all, even when the entire containing structure is translated, for example, a discussion (in multiple languages) of a phrase in a specific language or a localization concern. This is an editorial, not a technical decision, and must be flagged by the creator.

Here is the original, with the `@translate` attribute set to “no” to block translation of the onomatopoeia: [[JATSMLExamples]]

```
<p xml:lang="en" id="LRH-4011" lang-group="LRH-4011"
     lang-variant="original">Carrying the sack of Wheat, she trudged
     off to the distant mill. There she ordered the Wheat ground into
     beautiful white flour. When the miller brought her the flour
     she walked slowly back all the way to her own barnyard in her
     own <named-content content-type="quote" translate="no">
     picketty-pecketty</named-content> fashion.</p>
```

Here is the translation, following that instruction: [[JATSMLExamples]]

```
<p lang-group="LRH-4011"  xml:lang="cy"
     lang-variant="translation">Dan gario'r sach Gwenith,
     trampiodd i ffwrdd i'r felin bell. Yna, gofynnodd am falu'r
     Gwenith yn flawd gwyn, hyfryd. Pan ddaeth y melinwr â'r
     blawd iddi, cerddodd yn araf yn ei hôl i'w beudy ei hun, yn
     ei ffordd <named-content xml:lang="en"
     content-type="quote">picketty-pecketty</named-content>
     ei hun.</p>
```

## 6. Doesn't This Mechanism Entail a Lot of Overhead?

Well, it **can** entail a lot of overhead, or none at all; that depends on the needs and inclinations of the user. Users creating content can totally ignore it. Content creators who have multilingual content, *that they want to encode as multilingual content*, are “required” (which means encouraged) to include one metadata attribute (`@lang-grouping`) to flag that they are using this mechanism. In addition to the `@lang-grouping` attribute, users are expected to use as much, and only as much, of this infrastructure as is appropriate to their content and expected use. The minimum would be tagging all language groups with the `@lang-group` and `@xml:lang` attributes. Content receivers are not required to accept documents that use the multi-language mechanism, and can easily identify documents that use it tagging by looking for the `@lang-grouping` flag on the `<processing-meta>` element. Content receivers that do accept and process multilingual documents using the JATS multi-language mechanism can use the `@lang-grouping` flag to identify those documents that need this additional level of processing.

## 7. Documentation for All This

JATS is well documented. In fact, we have been told that JATS is **over documented** (we disagree). The official documentation of JATS is the JATS standard, which states the rules of the vocabulary. Like most standards, it does not contain explanations, advice to the user, or examples. In our opinion, the main value in the standard is the fact that it exists.

The JATS multi-language mechanism is clearly and helpfully described both in the JATS non-normative documentation (Tag Libraries) and in several articles and conference papers:

- The most complete introduction is the discussion called “Multiple Languages/scripts” inside the “Common Tagging Practice” section of each of the JATS Tag Libraries. This essay includes: discussion of the requirements for multiple language encoding, a description of the basic JATS 1.4 attribute and element mechanism, definitions and restrictions for all the multilingual attributes, an explanation of other JATS changes that were made to accommodate multilingual encoding, and numerous examples. [[JATSMultilang]]

- The “Attribute” Chapter of each JATS 1.4 Tag Library documents each of the multi-language attributes individually (with definition, values, and examples). [[JATSTaglib]]

- Encoding multilingual bibliographic references is described in “Multiple Language Citations” subsection of the “References” subsection of the “Common Tagging Practice” section of each of the JATS Tag Libraries. [[JATSMLCit]]

- Many additional multi-language attribute examples are provided by B. Tommie Usdin in her JATS-Con 2025 paper “Tagging multi-lingual documents in JATS 1.4: Some Examples”. [[JATSMLExamples]]

- There have been presentations on the JATS multi-language mechanism at JATS-Con ([[NewinJATS1.4], [ImproveMLJATCon], [JATSMLExamples]]), in the NISO Update series [[JATSUpdt]], and a paper about it was published in “Science Editing” [[ImproveML]].

For full technical details, examples, and discussions of usage we refer the reader to these sources.

## 8. Acknowledgements

The authors thank Bethan Tovey-Walsh for the translations into Welsh and her very helpful discussion of the issues in translating literary texts in general and onomatopoeia in specific.

## Bibliography

[JATSTaglib] National Center for Biotechnology Information (NCBI), National Library of Medicine (NLM): Journal Archiving and Interchange Tag Library NISO JATS Version 1.4 (ANSI/NISO Z39.96-2024). October 2024. Archiving: https://jats.nlm.nih.gov/archiving/tag-library/1.4/, Publishing: https://jats.nlm.nih.gov/publishing/tag-library/1.4/, Authoring: https://jats.nlm.nih.gov/articleauthoring/1.4/

[JATSMultilang] National Center for Biotechnology Information (NCBI), National Library of Medicine (NLM): Journal Archiving and Interchange Tag Library NISO JATS Version 1.4 (ANSI/NISO Z39.96-2024) Multiple Languages/scripts. October 2024. Archiving: https://jats.nlm.nih.gov/archiving/tag-library/1.4/chapter/tag-multi-lang-articles.html, Publishing: https://jats.nlm.nih.gov/publishing/tag-library/1.4/chapter/tag-multi-lang-articles.html, Authoring: https://jats.nlm.nih.gov/articleauthoring/tag-library/1.4/chapter/tag-multi-lang-articles.html

[JATSMLCit] National Center for Biotechnology Information (NCBI), National Library of Medicine (NLM): Journal Archiving and Interchange Tag Library NISO JATS Version 1.4 (ANSI/NISO Z39.96-2024) Multiple Language Citations. October 2024. Archiving: https://jats.nlm.nih.gov/archiving/tag-library/1.4/chapter/tag-cite-multi-lang.html"> [https://jats.nlm.nih.gov/archiving/tag-library/1.4/chapter/tag-cite-multi-lang.html], Publishing: https://jats.nlm.nih.gov/publishing/tag-library/1.4/chapter/tag-cite-multi-lang.html, Authoring: https://jats.nlm.nih.gov/articleauthoring/tag-library/1.4/chapter/tag-cite-multi-lang.html

[JATSStd] American National Standards Institute/National Information Standards Organization (ANSI/NISO): ANSI/NISO Z39.96-2024. JATS: Journal Article Tag Suite, version 1.4. Baltimore: National Information Standards Organization. https://groups.niso.org/higherlogic/ws/public/download/31415/ANSI-NISO-z39.96-2024.pdf

[HyTime] The Editors of ISO/IEC 10744:1992 2nd Edition: Charles Goldfarb, Information Management Consulting, Steven R. Newcomb, TechnoTeacher Inc., W. Eliot Kimber, ISOGEN International Corp., and Peter J. Newcomb, TechnoTeacher Inc. A Reader's Guide to the HyTime Standard. https://www.hytime.org/papers/htguide.html

[ITS2.0] Internationalization Tag Set (ITS) Version 2.0, W3C Recommendation 29 October 2013http://www.w3.org/TR/2013/REC-its20-20131029/

[NewinJATS1.4] D.A. Lapeyre: What’s New in JATS 1.4. In: Journal Article Tag Suite Conference (JATS-Con) Proceedings 2025 [Internet]. Bethesda (MD): National CentNational Center for Biotechnology Information (US); 2025. https://www.ncbi.nlm.nih.gov/books/NBK611601/

[ImproveML] Vincent Lizzie: Improving Journal Article Tag Suite for Multilingual Articles. Sci Ed. 2022; 9: 2:169–178. DOI:https://doi.org/10.6087/kcse.285. https://www.escienceediting.org/journal/view.php?number=291

[ImproveMLJATCon] Vincent Lizzi: Improving JATS for Multilingual Articles. Journal Article Tag Suite Conference (JATS-Con) Proceedings 2022 [Internet]. Bethesda (MD): National Center for Biotechnology Information (US); 2022. https://www.ncbi.nlm.nih.gov/books/NBK579699/

[JATSUpdt] Tommie Usdin: Open Teleconference: JATS (Journal Article Tag Suite Update, November 18, 2024. NISO. Audio recording (discussion with NISO’s Keondra Bailey), https://niso.org/events/2024/11/opentelecon-jats

[JATSMLExamples] B.T. Usdin: Tagging multi-lingual documents in JATS 1.4: Some Examples. In: Journal Article Tag Suite Conference (JATS-Con) Proceedings 2025 [Internet]. Bethesda (MD): National CentNational Center for Biotechnology Information (US); 2025. https://www.ncbi.nlm.nih.gov/books/NBK611124/

Markup UK

https://markupuk.org/