# 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 .