A Pilot Implementation of ixml

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

Abstract

Invisible XML (ixml) is a method for treating non-XML documents as if they were XML[7], [8], [9], [10], [11], enabling authors to write documents and data in a format they prefer while providing XML for processes that are more effective with XML content. By the time of the publication of this paper, it is anticipated that the official version of ixml [12] will have been announced by the ixml working group.

During the development of ixml, a pilot implementation was built in order to support decisions on the development of the notation, and to provide examples of the output ixml produces.

This paper describes the implementation, decisions taken, and how certain processes work, such as serialisation, and dealing with ambiguity, and ends by discussing future work to be done.

Keywords: ixml, markup, XML, implementation, notation design, grammars, parsing, earley

1. Invisible XML

Numbers are abstractions: you can't point to the number three, just three apples, or three sheep. Three is what those apples and sheep have in common.

You can represent a number in different ways, 3, III, 0011, 三 , ३, ৩, ૩, ੩, 〣, ೩, ๓, ໓, ༣, III, ൩, ၃, ፫, ꧓, trois, drie. You can concretise numbers as a length, a weight, a speed, a temperature.

But in the end, they are all the same *three*.

The idea behind ixml is that data too is an abstraction, which we are often obliged for different reasons to represent in some way or another. But in the end those representations are all of the same abstraction. It is worth noting that http recognises this too, with content negotiation around a URL: the URL represents a single resource; content negotiation allows the selection of a particular representation of that resource [6].

Ixml takes representations of data, typically with implicit structure, recognises that structure, and forms a representation where the structure is made explicit.

Some representations are weaker than others: they may not be able to faithfully represent all of the abstraction, and are therefore not reversible, but XML is probably the best available general notation for approaching the representation of any abstraction.

The intention behind ixml is to allow extracting abstractions from representations; of converting weaker representations of abstractions into more explicit representations, with XML therefore an excellent target for that.

2. Processing

An ixml processor takes a a document in a particular (textual) format, along with a description of that format, in the form of a grammar, and uses it to parse the document. This produces a structured parse tree of the document, which can then be processed in a number of ways, the primary one being serialization as XML.

Figure 1 illustrates this: the circle is the processor, a square represents a textual representation of a document, and a triangle a structured representation. The documents to the left and right of the processor are the same document, but in different representations.

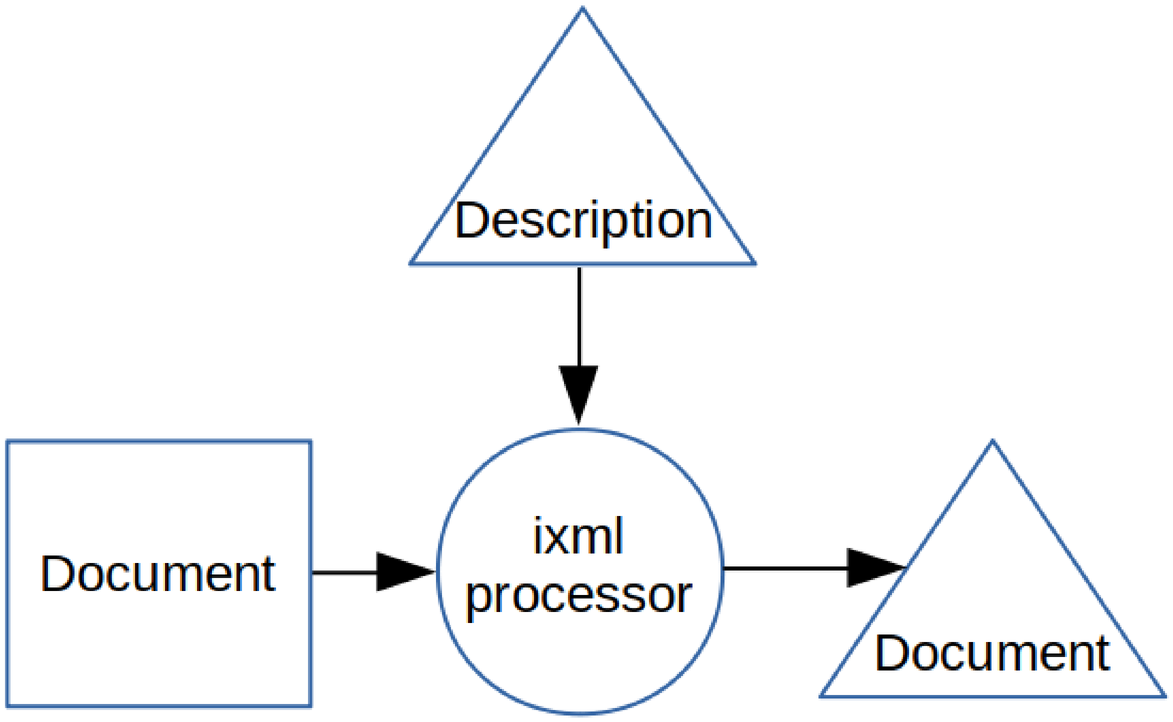

Figure 1. Fig. 1: ixml processing step

As you can see, the format description is drawn as a structured document. However, since it is normally supplied as an ixml document in textual form, this also has first to be processed, in exactly the same way, by the ixml processor, but using a description of the ixml format. This results in the structured version of the description. Figure 2 illustrates this. As you can see there is a also a presumed bootstrap stage that produces the initial structured version of the description of ixml itself, where the structure of ixml is initially presumed during the bootstrap.

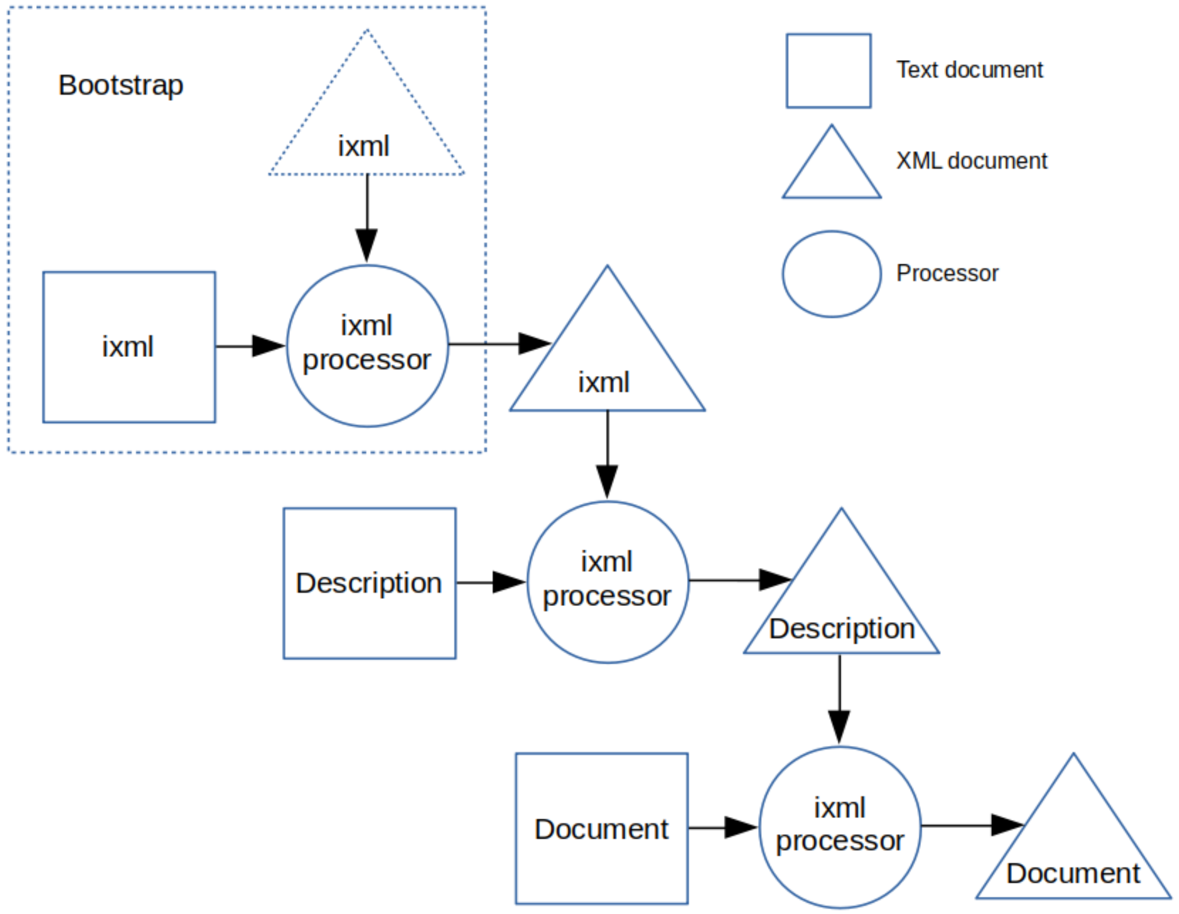

Figure 2. Fig. 2: The complete processing cycle

3. How ixml works

An ixml description is a grammar consisting of a series of rules. A rule consists of a name, and a number of alternatives separated by semicolons:

```
statement: assign; call; if; while.
```

Alternatives consist of a sequence of zero or more terminals and nonterminals separated by commas:

```
assign: id, ":=", expr.
expr: id; number.
```

Input that matches the grammar is parsed into a parse tree, which is then serialised as XML. So for input "`i:=0`", you would get

```
<statement><assign><id>i</id>:=<expr><number>0</number></expr></assign></
statement>
```

So-called *marks* can be added to rules to affect the serialization. Rules can be marked to be serialised as attributes:

```
assign: @id, ":=", expr.
```

which would give

```
<assign id="i">:=<expr><number>0</number></expr></assign>
```

terminals can be marked to be deleted:

```
assign: @id, -":=", expr.
```

which would give

```
<assign id="i"><expr><number>0</number></expr></assign>
```

as can nonterminals:

```
assign: @id, -":=", -expr.
```

which would eliminate the enclosing `<expr>` tags around its element content:

```
<assign id="i"><number>0</number></assign>
```

A recent addition to ixml are *insertions* which allow characters to appear in the serialisation that weren't in the input:

```
assign: @id, -":=", -expr, +";".
```

which would give

```
<assign id="i"><number>0</number>;</assign>
```

4. Implementation

4.1. Parsing algorithms

There are many known parsing algorithms, and most have restrictions of one sort or another on the classes of language that they recognise, and include restrictions on the languages, or the grammars, or both, for them to work.

For example, LL(1) grammars require that when a grammar rule has alternatives (as most do), like

```
sentence: a; b; c.
```

that the choice of whether to parse the sentence as an `a`, `b`, or `c` must be decidable by looking at most one token ahead. To achieve this, languages typically must go through an initial stage prior to parsing to split them into individual tokens, which in turn means that a grammar has to be accompanied by a description of how tokens are formed.

All this makes life difficult for the grammar writer, who must not only know the rules for the parsing algorithm, and how to apply them, but must write two descriptions, one for the grammar, and another for the tokens.

To avoid these problems, ixml requires implementations to use a general parsing algorithm, without extra restrictions, and without the need to specify tokens. Examples of such algorithms are [3], [14], [2], [5], and [4].

4.2. Parsing

The pilot implementation of ixml uses Earley as one of the earliest and best-known of the general parsing algorithms.

As pointed out in [9], Earley can be seen as a pseudo-parallel parsing algorithm, where when it has to parse a rule like

```
sentence: a; b; c.
```

it splits into three parallel sub-parsers to parse the three alternatives starting at the same point in the input. If a sub-parse fails at any point, it terminates without further ado; if it succeeds, it records its sub-parsetree(s), and terminates.

4.3. Serialisation

Once parsing is finished, what remains is a so called parse-forest, a collection of linked sub-parse-trees.

The first action is to see if the parse has been successful, by looking if there is a successful parse node for the root symbol that starts at the first character position and ends at the last. If so then serialisation can begin.

Serialisation is a question of doing a tree walk: non-deleted nonterminals are serialised to XML elements, deleted nonterminals just have their children serialised, non-deleted terminals (and insertions) are just output. There is an additional elaboration for nonterminals serialised as attributes, since they come before element content, and so for any non-deleted nonterminal, you have to do one walk for the attributes, and then one for the content.

Because of element deletions such as

```
assign: -target, -":=", expr.
target: @id.
```

since the element that the id attribute is ostensibly on is deleted, the attribute has to move up to the nearest non-deleted element:

```
<assign id="i"><expr>...</expr></assign>
```

so the tree walk for attributes has to look not only at the level of the (undeleted) element, but also recursively within deleted sub-elements.

4.4. Ambiguity

The parse may have been ambiguous: that is, it satisfied the rules of the grammar in more than one way. The serialisation tree-walk is not going to care about this, and as long as the parse has been successful it will produce a serialisation of one of the parses.

However, it is usually important that the consumer of the serialisation know that the serialisation is only one of the possible cases. To this end, ixml requires an `ixml:state="ambiguous"` attribute to be added to the root element of the serialisation to signal that fact. This involves a simple initial tree-walk to discover if any route from the top node is ambiguous.

5. The Pilot Implementation in Use

The pilot implementation was originally written to support the development of the language, to try out different approaches and solutions, as well as to generate example outputs to support papers on the language.

The primary aim at that stage was therefore speed of implementation and flexibility, and not to create an industrial-strength, top-speed implementation, since it was too early at that stage while the language was constantly being altered. Consequently it was written in an interpreted language with very-high-level data types, ABC [1], in about 500 lines of code for the bootstrap parser, and 700 for the full processor.

It was later also used to support an ixml tutorial [13]. After an earlier experience with a tutorial for a different language, where the tutees had to install software themselves to run the examples, it was decided for ixml instead to supply an online processor, where the example ixml grammar and its input were submitted to the processor, and the resulting XML would be returned. A problem was that the pilot implementation was non-reentrant, and so the solution was to store the grammars and input in files, and serve them one by one to the processor, while the server busy-waited for the result file.

6. Future Work

6.1. Serialising to memory

Although the processing diagram above suggests that ixml always serialises its input documents to XML, there are other options. In particular since the format-description document is used as input in the next step to parse the final document, it can be more efficient to serialise the format description straight to memory, into the form required by the parser for representing grammars. This also allows for some simplifications, in particular because the grammar for ixml will never produce ambiguous parses, and so doesn't need the two-passes otherwise necessary for serialisation. This of course also speeds up processing, since it eliminates one whole parsing phase.

6.2. Round-tripping

An ixml grammar can be seen as a function mapping one representation on to another. In simple cases, such as a grammar with no deletions and no attributes, mapping the output back to the input form is trivial, since it just involves concatenating the element contents. This is because the default is for all input characters to be copied to the output serialisation, and in simple cases like this all that happens is content gets enclosed by element tags to reveal the underlying structure. Remove the tags, and you have the input again.

Deletions complicate matters. If there are only element deletions, then it remains trivial, because all input characters are still in the output; only some element tags have been omitted, so concatenating element content still works.

However, with terminal deletions, characters are lost that have to be restored. To deal with this, we have to parse the serialisation using the grammar that produced it, with a similar parser to Earley, in order to discover which characters have been deleted.

To parse a (non-deleted) nonterminal, you must expect the start tag for that rule. For instance, for

```
statement: assign; call; if; while.
```

you expect

```
<statement>
```

and then initiate four parallel sub-parsers, one for each of the alternatives, which to succeed must also be followed by the terminating tag `</statement>`.

To parse a deleted nonterminal, you just initiate the sub-parsers.

To parse a (non-deleted) terminal, you must just expect the same string in the element content at the current point.

To parse a deleted terminal, you match zero characters, and insert the string in the parse. Interestingly enough, this is exactly how an insertion works when parsing in the other direction. And indeed to parse an insertion, you have to expect the characters in the string in the same way as a non-deleted terminal, but insert nothing in the parse, just like a deletion in the other direction.

The other challenge is dealing with attributes. Because of how attributes are placed on XML elements, and additionally due to deleted nonterminals as explained above, attribute content can appear earlier in the serialisation than the content that preceded them in the input. Therefore attributes have to be held in abeyance, as separate input streams, until the point in the parse where they appear in the grammar. Then the input must come from the serialisation of the attribute, and all sub-rules must be treated as if deleted, until the end of the rule forming the attribute, which by then must have consumed all of the input coming from the serialisation of the attribute.

The result of this process will be a (potentially ambiguous) parse tree.

For a serialisation like

```
<assign id="i"><number>0</number>;</assign>
```

being parsed against

```
assign: @id, -":=", -expr, +";".
```

you will get a parse tree like

```
<assign><id>i</id>:=<expr><number>0</number></expr></assign>
```

from which you can concatenate the element content to give "`i:=0`".

However, for a grammar that includes a rule like:

```
term: id; number; -"{", expr, -"}"; -"(", expr, -")".
```

where expressions may be bracketed with either {} style or () style brackets, both of these latter two alternatives, because of the deletions, will produce exactly the same serialisation, namely:

```
<term><expr>...</expr></term>
```

and so parsing the serialisation back we will get two successful parses:

```
<term>{<expr>...</expr>}</term>
```

and

```
<term>(<expr>...</expr>)</term>
```

in other words, an ambiguous parse. So, just as with serialisation, we have to choose one. In other words, because the serialisation throws away the information about which brackets were used, we can't guarantee to round-trip the serialisation perfectly.

Similarly, if a serialisation deletes all spaces or comments in the input, there is no way to recreate them. They are lost in the round-tripping.

6.3. Translating to other languages

The main shortcoming of the pilot implementation is that it is written in a programming language that doesn't fully support Unicode. While it is possible to parse Unicode documents with it, any feature of ixml that requires accessing *properties* of Unicode characters, such as hexadecimal representations, ranges, or character classes, can't be fully supported.

To this end, future work will include translating the implementation to one or more other languages.

7. Conclusion

Designing a notation requires many aspects to be taken into account simultaneously, such as usability, functionality, and ambiguity. Having an implementation that is easily modifiable during design of the notation is almost essential for good progress.

The current pilot implementation has served well, and while other implementations are now emerging, it will probably be retained for future design research.

References

[1] Leo Geurts et al.. *The ABC Programmer's Handbook*. Prentice-Hall 1990. 0-13-000027-2. https://www.cwi.nl/~steven/abc/programmers/handbook.html .

[2] Itiroo Sakai. *Syntax in universal translation*. In 1961 International Conference on Machine Translation of Languages and Applied Language Analysis 1961. 593–608. https://aclanthology.org/www.mt-archive.info/50/NPL-1961-Sakai.pdf .

[3] J. Earley. *An efficient context-free parsing algorithm*. In Communications of the ACM 13(2) February 1970. 94–102. 10.1145/362007.362035.

[4] ElizabethAdrian ScottJohnstone. *GLL Parsing*. In Electronic Notes in Theoretical Computer Science Volume 253.7 17 September 2010. 177-189. 10.1016/j.entcs.2010.08.041.

[5] Masaru Tomita. *Generalized LR Parsing*. Springer Science & Business Media 1991. 978-1-4615-4034-2. 10.1007/978-1-4615-4034-2.

[6] R. Fielding et al.. *Hypertext Transfer Protocol -- HTTP/1.1*. IETF 1999. https://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html .

[7] Steven Pemberton. *Invisible XML*. Proceedings of Balisage: The Markup Conference 2013 Balisage Series on Markup Technologies vol. 10 2013. 10.4242/ BalisageVol10.Pemberton01.

[8] Steven Pemberton. *Data Just Wants to Be Format-Neutral*. Proc. XML Prague 2016 2016. 109-120. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf .

[9] Steven Pemberton. *Parse Earley Parse Often: How to Parse Anything to XML*. In Proc. XML London 2016 2016. 120-126. http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120 .

[10] Steven Pemberton. *On the Descriptions of Data: The Usability of Notations*. In Proc. XML Prague 2018 2018. 143-159. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf .

[11] Steven Pemberton. *On the Specification of Invisible XML*. Proc.XML Prague 2019 pp 413-430. https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf#page=425 .

[12] Steven Pemberton. *Invisible XML Specification*. invisiblexml.org 2022. https://invisiblexml.org/ixml-specification.html .

[13] Steven Pemberton. *Hands On ixml*. Declarative Amsterdam 2021. https://declarative.amsterdam/show?page=da-tutorial-ixml-2021 .

[14] S.H. Unger. *A global parser for context-free phrase structure grammars*. In Communications of the ACM 11(4) April 1968. 240–247. 10.1145/362991.363001.

8. Postscript

In this author's experience, the hardest part of getting an article into Docbook format (the format used by this conference) is getting the bibliography right. To this end, the above bibliography was produced with the help of ixml. For instance, the text

```
[spec] Steven Pemberton (ed.), Invisible XML Specification,
invisiblexml.org, 2022, https://invisiblexml.org/ixml-
specification.html
```

was processed by an ixml grammar whose top-level rules were

```
bibliography: biblioentry+.
biblioentry: abbrev, (author; editor), -", ", title, -", ", publisher,
-", ", pubdate, -", ", (artpagenums, -", ")?, (bibliomisc;
biblioid)**-", ", -#a.
```

yielding

```
<biblioentry>
 <abbrev>spec</abbrev>
 <editor>
 <personname>
       <firstname>Steven</firstname>
       <surname>Pemberton</surname>
 </personname>
 </editor>
 <title>Invisible XML Specification</title>
 <publisher>invisiblexml.org</publisher>
 <pubdate>2022</pubdate>
 <bibliomisc>
 <link xlink:href='https://invisiblexml.org/ixml-specification.html'/>
 </bibliomisc>
</biblioentry>
```

which can further be tweaked by hand.