Round-tripping Invisible XML

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

Abstract

Invisible XML takes textual documents where the structure is implicit and produces documents with the structure made explicit. This paper addresses the question of the extent to which it is possible to recreate the original textual document from its structured version, how it can be done, and what if any the ramifications are for ixml.

Keywords: ixml, structured data, parsing, XML, round-tripping, grammar transformation, serialisation

1. Contents

2. Introduction

Invisible XML [1] is a language and process that takes linear textual input, recognises the implicit structure in the input, and converts it to an equivalent structured XML output. It does this by parsing the input using a grammar describing the format of the input document, and serialising the resultant parse-tree as XML, using extra information in the grammar to drive the serialisation.

If that were all it did, then round-tripping the XML back to text would be trivial: it would simply be a case of concatenating the text nodes of the XML, and you'd be done.

However, there are issues with regards to ixml serialisation:

- input characters may be deleted from the parse-tree on serialisation;
- extra characters that weren't in the input may be inserted;
- some parse-tree nodes may be serialised as attributes rather than elements, causing a reordering of the input text, since attributes appear before element content.

As hinted at in earlier papers on ixml [2], [3], round-tripping could be achieved by having a special-purpose general parser which attempts to recreate a parse-tree that could have produced the serialisation, and then concatenating the resulting text nodes.

This paper takes a different approach: by transforming the input grammar into a grammar that represents all possible serialisations of the input grammar, it can use the same parser as used by ixml, with some small additions, to parse the serialisation back into a parse-tree that would have produced that serialisation.

This raises a number of technical issues similar to the normal ixml process, in particular what to do with ambiguity, where a serialisation could have been produced by more than one input.

3. Round-tripping

The term round-tripping is normally considered to be the process of recreating the original document that produced the output you have [4]. For several reasons discussed below, except in limited cases, it is not possible to recreate a character-perfect original document from the ixml output. This is partly under the grammar author's control, which is to say, it is possible to write an ixml grammar that can be perfectly round-tripped. On the other hand, in the other direction round-tripping is always possible: it is always possible to construct from the ixml *output* an input that would create exactly the same output again.

So for the purposes of this paper, our definition of round-tripping is not "create an identical input to the one that created this output", but:

Create an input that would produce the identical output.

As an example, an ixml grammar that reads a program in a programming language, deleting comments and spurious whitespace, on round-tripping would of course no longer contain the comments and extra whitespace: they were deleted and didn't appear in the structured version. However the round-tripping would still produce effectively the same program.

4. The Problem Space

Let's take the simplest case. Here is a simple grammar for dates:

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

Given as input

```
30/06/2024
```

this will produce as output:

```
<date>
   <day>30</day>/
   <month>06</month>/
   <year>2024</year>
</date>
```

As you can see, all characters in the input are preserved in the output, in order. The only difference is that tags have been placed around the characters to indicate the structure.

So to round-trip this, all that is necessary is to concatenate the text nodes of the XML, to give the input that created the output:

```
30/06/2024
```

This is what is meant above by "it is possible to write an ixml grammar that can be perfectly round-tripped": if the grammar deletes no characters, and only uses elements, round-tripping is trivial.

The fact that some tags have been suppressed in the output (namely the `<d>` elements, due to the "-" before the rule for `d`) has no effect, since we are only interested in characters.

However, ixml has facilities to control the serialisation format. For instance, the slash characters "/" in the input are there only as punctuation to separate the different parts, but play no role in the output, and so can be deleted from the serialisation:

```
date: day, -"/", month, -"/", year.
```

giving

```
<date>
   <day>30</day>
   <month>06</month>
   <year>2024</year>
</date>
```

Now we can no longer just concatenate the text nodes to get back to the input.

Similarly, characters can be inserted into the serialisation. Suppose the input date format only had two digits for the year, but the output serialisation required four. You can write this:

```
date: day, -"/", month, -"/", year.
day: d, d.
month: d, d.
year: +"20", d, d.
-d: ["0"-"9"].
```

which for the input `30/06/24` would give:

```
<date>
   <day>30</day>
   <month>06</month>
   <year>2024</year>
</date>
```

Finally, elements can be serialised as attributes:

```
date: day, -"/", month, -"/", year.
day: d, d.
month: d, d.
@year: +"20", d, d.
-d: ["0"-"9"].
```

which gives

```
<date year='2024'>
   <day>30</day>
   <month>06</month>
</date>
```

This reorders the input, since attributes are serialised before elements.

These issues need to be addressed in order to enable round-tripping of content in the general case.

5. Grammar Transformation

As already pointed out, earlier papers have suggested using a special-purpose parser to deal with round-tripping, but this paper proposes a different approach: transforming the input ixml grammar into a different ixml grammar that recognises the output format, and then using the same parser that ixml already uses.

As an example, for the first grammar above

```
date: day, "/", month, "/", year.
day: d, d.
month: d, d.
```

```
year: d, d, d, d.
-d: ["0"-"9"].
```

a grammar can be generated that recognises the serialisation:

```
date: -"<date>", day, "/", month, "/", year, -"</date>".
-day: -"<day>", d, d, -"</day>".
-month: -"<month>", d, d, -"</month>".
-year: -"<year>", d, d, d, d, -"</year>".
-d: ["0"-"9"].
```

Using this grammar and the regular ixml parser to parse the output XML, gives:

```
<date>30/06/2024</date>
```

(We have glossed over the issue of extra inserted whitespace for pretty printing, which we will discuss later; we will also detail later reasons why the root element remains).

This leads immediately to the second example. where everything is the same except the rule:

```
date: day, -"/", month, -"/", year.
```

with a transformed equivalent rule:

```
date: -"<date>", day, +"/", month, +"/", year, -"</date>".
```

which similarly generates the same output

```
<date>30/06/2024</date>
```

And finally (for now), the version with added output characters

```
date: day, -"/", month, -"/", year.
day: d, d.
month: d, d.
year: +"20", d, d.
-d: ["0"-"9"].
```

with transformation:

```
date: -"<date>", day, +"/", month, +"/", year, -"</date>".
-day: -"<day>", d, d, -"</day>".
-month: -"<month>", d, d, -"</month>".
-year: -"<year>", -"20", d, d, -"</year>".
-d: ["0"-"9"].
```

giving as output:

```
<date>30/06/24</date>
```

So from these three examples we can see some patterns for the transformations emerging:

- non-hidden rules except for the root become hidden rules in the transformation, and start and end with its tags, both marked as deleted;
- hidden rules remain hidden, and no tags are recognised;
- in alternatives:
  - nonterminals remain the same;
  - regular terminals stay the same;
  - deleted terminals become inserted terminals;
  - inserted terminals become deleted terminals.

6. Dealing with Attributes

The main problem left arises with attributes, which turn up at a different position in the serialisation. For instance the grammar

```
date: day, -"/", month, -"/", year.
day: d, d.
month: d, d.
@year: +"20", d, d.
-d: ["0"-"9"].
```

which generates the output

```
<date year='2024'>
   <day>30</day>
   <month>06</month>
</date>
```

which can be *recognised* with

```
date: -"<date", year, -">", day, +"/", month, +"/", -"</date>".
-day: -"<day>", d, d, -"</day>".
-month:-"<month>", d, d, -"</month>".
-year: -" year='", -"20", d, d, -"'".
-d: ["0"-"9"].
```

but would produce on output

```
<date>2430/06/</date>
```

In other words, while it can recognise the attribute, it turns up in the wrong place for the round-tripping, and so a mechanism is needed for putting it back in the right place.

In normal ixml processing, when serialising to XML, the ixml grammar identifies the place in the input where the eventual attribute is, and implicitly identifies the place where the attribute must be serialised on output, namely the nearest ancestor element that isn't hidden. This is done with a two-pass algorithm: when serialising an element, children of the element, and of hidden children elements are traversed first to find rules marked as attributes, which are then serialised before the element children.

To reverse this process, we must recognise the attribute at its implicit position in the input, and *explicitly* identify in the grammar the place where it needs to be serialised; this will always be later in the output than where the attribute was found. We can do this by defining two extra *marks*: one to indicate that the attribute as parsed should not be serialised at that position, and another to indicate the position where it should be serialised.

This is comparable to how terminals are inserted and deleted on serialisation: `-"abc"` means "*parse the string but serialise nothing"*, while `+"abc"` means "*parse nothing, but serialise the string"*. Unfortunately, "-" has already been assigned a different role for nonterminals on serialisation, but "+" is free to use with the same meaning.

So using the mark "*" to mean "*parse the input, but serialise nothing*", and "+" to mean "*parse nothing but serialise the node of this name from earlier in the tree marked with a "*"*", we can specify:

```
date: -"<date", *year, -">", day, +"/", month, +"/", +year, -"</date>".
-day: -"<day>", d, d, -"</day>".
-month: -"<month>", d, d, -"</month>".
-year: -" year='", -"20", d, d -"'".
-d: ["0"-"9"].
```

This now correctly produces

```
<date>30/06/24</date>
```

Note that these are not changes to the ixml language itself, but only internal additions: the transformations are done on internal representations of the grammar, and not on external representations (but see the section on future work).

7. Strict and Permissive Grammars

There are two (non-normative) terms used to describe two major applications of ixml grammars: *strict* and *permissive*.

If it is certain that the input being processed with ixml is correct, the grammar can be laxer in what it accepts. The date grammar is a good example of this. If dates are only being recognised, and not checked for correctness, then "`d, d`" is a perfectly good pattern for recognising the day number. Even though this would also recognise `32` or `99`, since such dates never occur, you don't have to worry about them. Such grammars are referred to as *permissive*.

On the other hand if the input might not be correct, then the grammar needs to be stricter. With dates again, you must then only accept single digits in the range 1 to 9, and double digits up to 31. Another example is the grammar for ixml itself, which clearly has to be strict.

For round-tripping we can assume the input is correct, and so the round-tripping grammar can be permissive. For instance, if we take a grammar for a programming language, that contains the rule

```
function-call: name, -"(", parameters, -")".
```

If `parameters` can be empty (for instance `now()`), then the output for that could be either

```
<function-call><name>now</name><parameters></parameters></function-call>
```

or

```
<function-call><name>now</name><parameters/></function-call>
```

and we neeed to generate grammar alternatives to recognise both forms.

But if `parameters` cannot be empty, then we wouldn't need to generate an alternative for the second case, since it will never occur. However, exactly since it will never occur, we don't have to worry about it being in the grammar anyway: since it will never occur, that alternative will never be matched.

Similarly for attributes. If we have a hypothetical grammar that contains something like:

```
element: a, b, c, body.
@a: acontent.
@b: bcontent.
@c: ccontent.
acontent: ...
bcontent: ...
ccontent: ...
body: ...
```

if we were being strict, we might want to generate alternatives that recognise that `a`, `b`, and `c` could appear in any order:

```
-element: -"<element", (abc; acb; bac; bca; cab; cba), -">", body, -"</
element>".
abc: a, b, c.
acb: a, c, b.
```

etc.

However, we can actually produce a permissive rule like:

```
-element: -"<element", (a; b; c)*, -">", body, -"</element>".
```

```
-a: " a='", -acontent, "'".
-b: " b='", -bcontent, "'".
-c: " c='", -ccontent, "'".
```

since this specifies that the element has three attributes, without requiring them to be in any particular order. While this grammar also permits `<element a="xxx"`

`a="yyy">` , this is not a problem, since it will never actually occur in the XML we are required to process.

8. Syntactic Equivalences

ixml adds a number of extensions to its grammar-description language, to make life easier for the author, and to make grammars more readable, extensions such as `?` for options, `*`, `**`, `+`, and `++` for repetition, and `(` and `)` for grouping.

However, while these extensions add expressiveness to the language, they don't add any recognition power: as pointed out in the ixml specification, they can all be straightforwardly transformed into grammars without the extensions, while recognising the same language, and in fact all implementations do this transformation as a form of code generation when processing the language.

As a consequence, the round-tripping process doesn't have to take the syntactic extensions into account; it can either assume the initial transformations have alreadt been done, or do them itself; either way it only has to transform simplified grammar rules that contain none of the extensions.

9. Loss of Information

As pointed out earlier, some information in the source document can be lost when transforming to XML. If a programming language is being recognised, it might be decided not to include nodes for comments in the resulting XML. So on round-tripping, those comments will not reappear in the document. But this is not a problem, since our aim is just to produce a text document that would result in the same XML serialisation.

However, there are some constructs where a decision has to be made, principally because of *inclusions* and *exclusions*.

An inclusion is a ixml construct that allows any character from a set of characters to be matched. We have seen them already in the dates example: `["0"-"9"]` matches any single digit. The transformation of such an inclusion remains the same: if the input has a single digit, the output will have the same single digit, and so we recognise it in the same way. The problem comes with deletions. For strings `-"/"`becomes `+"/"` on transformation: a deleted slash on input becomes an inserted slash on round-tripping. However, with a deleted inclusion, such as `-["0"-"9"]`, all we know is that a digit was in the input and deleted, but we don't know which.

There are two options here, both with the same effect. Either transform `-["0"-"9"]` by taking a character from the allowable set (for instance the first), and transform to `+"0"`, or update the serialisation process to allow the construct `+["0"-"9"]`, though with the same semantic of outputting the first character from the set.

Either of these still match the requirement that the resulting round-trip would produce the identical serialisation.

The case for exclusions is slightly harder. A construct such as `-~["0"-"9"]` deletes any single character that *isn't* a digit. In this case a character in the *complement* of the set has to be chosen.

10. Ambiguity

In regular ixml processing it can occur that an input document matches the grammar in more than one way. The ixml processor is required to only serialise one of the possible matches, but must also report that the result is ambiguous: that this serialisation is only one of the possible interpretations of the input.

In a similar way, round-tripping can also be ambiguous, in this case meaning "*this is only one of the possible strings that can produce the same serialisation*". In fact, many ixml grammars that are not ambiguous will produce ambiguous round-trips. As a simple example, if symbols on the input may be separated by one or more spaces, that are then deleted on serialisation:

```
input: sym**spaces.
spaces: -" "+.
```

then on round-tripping any number of spaces would be acceptable. This can best be solved by producing the minimum acceptable on round-tripping: one for "+", and zero for "*".

In a similar way:

```
-optional-a: -"a"?.
```

is ambiguous, because nothing is produced in the output, so we don't know if an "a" was present or not in the input. In this case, either option is acceptable, though the shorter case is probably preferable.

Ambiguity is marked in ixml by adding an attribute to the root element. This is why it is necessary to retain the root element on output of the round-trip: the output is (flat) XML anyway, and we need somewhere to report ambiguity.

11. Inserted Layout

Some implementations may produce a "pretty printed" serialisation of the XML, with added newlines and indentation. While this could be added to the grammar productions in the transformed grammar, it adds a danger of extra ambiguity, since elements may already have whitespace in them as part of the serialisation. For that reason it is better to round-trip XML without added whitespace.

12. Future Work

12.1. Similarities between Serialisation and Transformation

It is worth remarking that the code to produce the transformed grammars is very similar to the code used for serialisation. This should be unsurprising, since both processes walk the parse-tree, and while one outputs the serialisation, the other outputs a grammar that recognises that serialisation. Consequently the two processes are almost isomorphic.

This raises the question: what happens if you take the transformed grammar, and *transform it again*, what does it produce? Well, it round-trips the round-trip: it produces a serialisation that would have produced the round-tripped text. In other words, it serialises the input as XML.

This already works in basic cases, for instance, round-tripping twice the original example in this paper, you get

```
&lt;date>&lt;day>30&lt;/day>&lt;month>06&lt;/month>&lt;year>2024&lt;/
year>&lt;/date>
```

(The `&lt;`s are because it is outputting text that represents XML, but not XML itself. For readability, here is that output with the `&lt;`s expanded:)

```
<date><day>30</day><month>06</month><year>2024</year></date>
```

The corollary of this is that the serialisation part of an ixml processor could be greatly simplified, only having to output characters, and not having to worry about elements and attributes. Serialisation could be then done by twice transforming the input grammar, and using that instead.

Transformations are currently only partially isomorphic, since there is no reverse operation for the `*` mark, and the proposal mentioned above of allowing the mark + on inclusions and exclusions would need to be adopted to prevent loss of information in the grammar on transformation.

A second corollary however is that the ixml processor is now unbound from XML, and could be used to produce other serialisations in a fairly straightforward way.

12.2. Possible Additions to ixml

Although the additions to ixml to enable round-tripping were added to the internal form of ixml, and not the external form, if these facilities were to be made available to users as well, some marks would be needed to be added to the language. These are:

- Parse and don't serialise, notated with * in this paper;
- Parse nothing, and serialise a node from elsewhere, notated with + in this paper, but probably needing to be more general than presented here;

- Flatten a node: undo the recognised structuring by serialising the node as if it were the content of an attribute.

13. Conclusion

Contrary to expectations, it is possible to round-trip the ixml process, as long as you properly define what is understood by round-tripping, and slightly adapt ixml serialisation. This technique is very simple, and not only that, actually allows the ixml processor to be simplified, and to some extent generalised. An added advantage is that, with some work that still has to be done, the process is entirely reversible.

References

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

[2] Steven Pemberton. *Invisible XML*. Proceedings of Balisage: The Markup Conference 2013 vol. 10. 2013. 10.4242/BalisageVol10.Pemberton01. http://www.balisage.net/Proceedings/vol10/html/Pemberton01/BalisageVol10-Pemberton01.html..

[3] Steven Pemberton. *A Pilot Implementation of ixml*. Proc. XML Prague 2022. 2022. 41-50. 978-80-907787-0-2 (pdf). https://archive.xmlprague.cz/2022/files/xmlprague-2022-proceedings.pdf#page=51 .

[4] Wikipedia editors. *Round-trip format conversion*. Wikipedia. 2024. https://en.wikipedia.org/wiki/Round-trip_format_conversion .