Optimizing XML for Comparison and Change

Nigel Whitaker

Robin La Fontaine

DELTAXML

Conference Paper: XML London

ABSTRACT

Almost every user of XML will, at some stage, need to do some form of comparison between different versions of their XML data or document. This could be because it is necessary to review the changes that have been made, or to check that the output from one version of some software is the same as the previous version, or to find changes so that they can then be processed in some other way.

Designers of XML formats often do not consider this requirement for comparison when designing an XML format. However, if this is taken into account at the design stage then it can make the usefulness of the XML greater and at the same time reduce the cost of developing software. This paper outlines some of the issues that can usefully be considered by XML designers to make comparison and change easier to handle. The design issues will also be of interest to any user of XML to provide a better understanding of processing change in XML.

CONTENTS

CONTENTS

Introduction

It is possible to achieve better comparison of XML if you have some knowledge of the data or format being processed. At DeltaXML we have produced specific comparison products for formats such as DocBook and DITA and we often help our customers using their own in-house data and document formats. When doing this work we've often said to ourselves "if only they would have done this... it would make their life so much easier". This paper is a review of some of these issues and recommendations - it could be called a 'wishlist' or 'manifesto' for comparison. Some of this is just 'good advice' that extends beyond comparison per-se. Taking some or all of this advice when designing XML formats will make comparison and other XML processing tools work better out of the box and with less configuration.

We consider comparison to be a problem of identifying similarity, aligning information, and then represent the changes so that they can be processed in various ways. Some data formats have built in mechanisms to describe change, for example the status attribute in DITA, similarly @revisionflag in DocBook and the <ins> and <del> elements in HTML. Where these facilities are provided it often makes sense to make a comparison tool that takes two inputs and produce a result which uses the facilities and is also a valid XML result. This goal sometimes conflicts with the design of the schema and some of these issues will be addressed later.

Use a DTD or Schema for Whitespace Handling

Consider **Figure 1, "Added data"** where a new (phone) element has been added to some contact data. The inexperienced user says "I've added an element, it has three children.". We check if a DTD is used and when not finding one, we reply (perhaps a little pedantically) with: "No you haven't. You've added two nodes, an element and some text (which is a newline and spaces). The element contains seven children - three elements and four text nodes.".

Figure 1. Added data

```
<contact>
   ...
   <phone type="work">
      <countryCode>44</countryCode>
      <areaCode>020</areaCode>
      <local>7234 5678</local>
   </phone>
</contact>
```

Of course, it is possible to make comparison software remove the extra indentation whitespace and provide some configuration options. But this is delegating responsibility to the user, who may not have as good an understanding of the data as the developers.

As a user of a comparison tool - you may see changes that you don't care about and are a nuisance - the four added text nodes in the example above. There are other less obvious implications that we should also mention including performance and alignment.

Most comparison algorithm implementations are similar to those of XSLT transformation, in that you generally need to store the inputs in memory to navigate around them. Storing those extra tree nodes for the unnecessary whitespace affects both performance (it takes time to load and process them), but more importantly the heap space requirements. If you are processing data like that above, it's quite possible to halve the heap memory size needed for comparison with proper whitespace handling.

Many comparison algorithms use *Longest Common SubSequence* (or *SubString*) optimization techniques [1]. These work best (give good results and have good performance) when there are reasonably long sequences of similar or identical content. When they are faced with XML data like that above, but where one input has an extra indentation whitespace node for each element and the other file does not (perhaps its a single line of XML without whitespace, which when loaded in an editor makes you go to immediately to the indent button), it is almost a nightmare scenario. The whitespace nodes break-up the sequences of element nodes that can match, furthermore the whitespace nodes are all identical (the same newline and number of spaces giving the same level of indentation) and match each other. This kind of data will often mismatch and is also slow to process.

So what is proper whitespace handling? It is possible to remove the indentation whitespace above by pre-processing the data, or having a post-process which ignores or removes the purely whitespace changes that are identified. But by far the best way of dealing with whitespace is to use a DTD or schema so that the parser can differentiate between element-only and mixed content [4]. When they are used (and associated with the data using a DOCTYPE or @xsi:schemaLocation), parsers such as Apache Xerces can use the ignoreableWhitespace callback method; comparison and other tools then know they can safely ignore that type of content.

Using Schemas and DTDs Consistently

We hope the previous section has provided a convincing argument to use a DTD or schema. However, if you go down this route it is worth remembering that using a DTD or schema has a number of implications. In addition to controlling whitespace and use for validity checking they are also used for 'Infoset Augmentation'.

Infoset Augmentation means adding data from DTD or schema to the resulting parsed representation. It is often used to specify values of attributes, for example that a table by default will have a 1 pixel border. It is also, more controversially, used to provide a default namespace to xhtml data. While it is possible in some cases to determine if data was provided by augmentation, we would encourage instead that DTD DocTypes and schema association be used consistently. This will avoid spurious attribute change that is often confusing to the user ("I can't see that in either of my inputs") and in the case of xhtml, avoid the issues around an body element not matching or aligning with an xhtml:body element.

We have recently been working on a comparison tool for a documentation standard. That standard included reference material for each element, in terms of a DTD grammar. It also included, as a download, a large set of modular DTD/entity files and a related set of W3C XML Schema files (.xsd), but nowhere in the standard did it mention how XML instance files were meant to use the DTD or xsd files; no mention of DOCTYPE requirements or use of schemaInstance, or statements about processing expectations. Implementers then chose whether to use a DOCTYPE or not. We are then faced with comparing mixed files and have to deal with the differences due to augmentation between the two inputs. If you provide DTD or xsd files as part of a standard or format definition and they are used for augmentation, then a little guidance to implementers on how you expect them to be used would sometimes be appreciated!

Use Appropriate Data Types

There's another benefit to using a DTD or schema, particularly for data-orientated XML. A DTD provides a limited form of data typing; attributes can be declared of type ID which constrains uniqueness, and whitespace normalization for CDATA attributes is different to that for other types of attribute. A richer form of data-typing is provided by W3C and RelaxNG schema languages. These use the XML Schema datatypes [2] to describe the types of elements and attributes. This information can then be used for normalization purposes or post- processing to remove unexpected change. For example, you may consider these timestamped elements to be equivalent: `<element time='2013-03-14T14:35:00Z'>` and `<element time="Thu 14 Mar 2013 14:35:00 GMT">`.

When using floating point (or 'double') numbers most developers programming in Java or C# are warned about doing direct comparison operations and told to use epsilon or 'units of least precision' (ULPs). Similar consideration should be given to XML data values, whether serialized as characters in an XML file or when loaded into an in-memory tree.

It's often simpler to think of all of the attribute and other datatypes as strings particularly when developing both reader and writer software. However, proper definition of the datatypes has benefits when for comparison and also for other more general forms of XML processing such as XSLT 2.0 transformation.

Consider using xml:space

We have discussed how an XML parser's whitespace handling is affected by the use of a DTD or schema. The `xml:space` attribute also has an effect on whitespace handling, but not within the parser. Instead it is used to control how downstream applications handle whitespace. Think of `xml:space="preserve"` as a *hands off my whitespace* instruction!

It is often needed because many text processing applications of XML will normalize text prior to presentation, collapsing sequences of spaces to a single space, normalizing between line breaks and spaces and fitting or breaking lines to fit the page or display width. There is an expectation that comparison tools do likewise, for example, not reporting where two spaces in one document correspond to three spaces in another. Therefore our default processing of textual data involves a whitespace normalization process. It is good practice to then provide `xml:space` support to users who wish to avoid this normalization.

The examples given in the XML spec for `xml:space` are of the `poem` and `pre` element. Many XML specifications then allow use of `xml:space` on various elements and this gives the user the opportunity to turn-off the default behaviour described above. We would suggest that grammar designers *sprinkle* the `xml:space` attribute around their text or mixed content whenever possible independently of whether they fix or default its behaviour on elements such as `pre`. This allows the user to refine the space handling on a per instance basis.

Consider Using xml:lang

Users of comparison tools like to see changes at fine granularity, they generally don't want to see which text nodes in an XDM tree have changed, but rather which 'words' have changed. In order to do this text is usually segmented or broken into smaller chunks for alignment and comparison. This process is harder than it initially appears. The concept of a word varies significantly in different languages and character systems. In Latin/ European alphabets words are often delimited by spaces or punctuation whereas in eastern scripts 'words' are often represented by a single glyph or Unicode character.

Software is available to do this kind of text segmentation (the Java BreakIterator classes, or the widely used ICU [3] library which supports better internationalization). However in order to do its job properly it needs to be told which language is being used. This is where `xml:lang` is very useful. It is a small part of the XML specification [4], but often bypassed. Please consider adding it to your DTD or schema even if you don't envisage having multiple languages in a single XML file. If you add `xml:lang` to the root element, then it is possible for software to provide a default value, even if the user does not. This could perhaps be based on some user preferences, or computer locale settings.

Using `xml:lang` has benefits beyond comparison. Consider the case of single, double or triple clicking a range of text in an editor. The behaviour of these actions is also something that is typically language dependant and could utilise this attribute. Another useful general benefit, discovered when preparing this paper, is that `xml:lang` can be used to control spell checking dictionaries.

Data Ordering Issues

Some forms of data have an associated order and others do not. We will use a contact record as an example, as shown in **Figure 2, "Mixed ordered and orderless contact data"**. In this example the order of the `addressLine` elements is significant for postal delivery. However users may not care about which order the various phone elements appear.

Figure 2. Mixed ordered and orderless contact data

```
<contact>
   <name>John Smith</name>
   <addressLine>25 Green Lane</addressline>
   <addressLine>Bloomsbury</addressline>
   <addressLine>London</addressline>
   <addressLine>UK</addressline>
   <postcode>W1 2AA</postcode>
   <phone type="office">+44 20 1234 5678</
   phone>
   <phone type="fax">+44 20 1234 5680</
   phone>
   <phone type="mobile">+44 7123 123456</
   phone>
</contact>
</contact
```

Our comparison algorithms, and we suspect most others, are suited to either ordered/list based processing, perhaps optimizing a metric such as edit distance, or are suited to set-based or 'orderless' processing perhaps using hashing techniques and Map based data structures. Mixing these algorithms together so as to process the example above increases complexity enormously. Instead we prefer to treat all of the children of an element as either ordered or orderless and use the algorithms separately.

So, for the above data, rather than having a DTD like this :

```
<!ELEMENT contact
   (name, addressLine*, postcode, phone*)>
```

we prefer to use something like this:

```
<!ELEMENT contact (name, address, phone*)>
<!ELEMENT address (addressLine*,
postcode)>
```

or perhaps this:

```
<!ELEMENT contact
   (name, addressLine*, postcode,
   phoneNumbers)>
<!ELEMENT phoneNumbers (phone*)>
```

Introducing these grouping elements to the contact data makes it possible to subdivide the ordered and orderless content. It also allows us to attach attributes to them to control how the comparison is then performed at a given level of the tree. These attributes are easy to add using XSLT if there is an XPath that corresponds to an orderless container. It is also possible to specify, and to some extent document, the orderless nature of the element in the DTD, for example:

```
<!ATTLIST phoneNumbers deltaxml:ordered
(true|false) "false")>
```

Ids and Uniqueness in XML

The XML specification allows attributes to be declared of type ID or IDREF. One use-case of this facility is for cross- referencing within a document. For example, a user inserts a table into a document and gives it an id, for example `<table xml:id="forecast">`, and when talking about the table would later write: `... In <xref linkend="forecast"/> we describe the..........` The assumption being that the processing system would put replace the `xref` with an appropriate reference.

The XML specification requires that such ids are unique and this is supported by the editing and processing applications. For comparison and change control we would like to recommend another property that such ids should have, is that of persistence. The cross reference is in some respect a cross-tree pointer in XML. That's fine, but it can also change and when it does we are faced with the problem of working out if the user has changed what is being pointed to, or if the thing being pointed to has changed. Perhaps it is the same thing, but it has also changed slightly? Working out what's happened gets very difficult in these cases. We would recommend that if you write software to read and write such content then as well as considering id uniqueness please also consider persistence through read/ write cycles.

We've seen some examples of content which is converted from a word-processor format into XML markup where sections, tables etc are automatically numbered with a scheme such as "sect1_1, sect1_2 .. ". From a comparison perspective this is almost next to being useless, the same information is usually extractable using xsl:number or count(preceding-sibling::*) type operations. When the user creates a cross-reference please don't change the user-provided text. Adding artificial ids is usually a hindrence for comparison, particularly when a new chapter or section is interspersed into the content and causes subsequent elements to be renumbered differently. Finally we would suggest that schema designers do not make id attributes REQUIRED so that application developers and users do not have to come up with, probably artificial, numbering schemes.

The example we have shown above is a good use-case for XML ids. We don't recall many others. There is a danger of id/xml:id mis-use. Consider the case of a data file with National Insurance (NI) numbers. Don't be tempted to declare these of type ID/xs:ID because NI numbers are considered unique. Sooner or later you may need to add vehicle registration numbers of some other similar unique identifier and then face the problem that ID/xs:ID provides a single global namespace. Consider using schematron or perhaps XSD 1.1 assertions to specify uniqueness, they tend to be more flexible and also extensible for future needs.

Grammar Design Issues

When generating a comparison output, making use of the facilities provided by the data or document format being processed to describe the changes is useful as it allows users to make use of existing publication and other software available for that format. If that software validates its input then the comparison result should ideally also be valid. However there are some design issues that make this quite hard. We will illustrate this type of issue using DITA.

The DITA task specialization has a DTD model essentially similar to:

```
<!ELEMENT taskbody
  ((steps | steps-unordered), result)>
```

Using the status attributes we normally say whether an element has been added or deleted and this can then be styled appropriately with red or green colouring or strike- through. When one element is changed to another we can normally represent this as one being deleted and another being added. However the design of the DITA grammar precludes this when steps is changed to steps- unordered or vice-versa.

We can see why the designers thought that there should only be a single sequence of steps in a task. But from our perspective its difficult to represent this type of change in standard DITA. We currently provide a parameter in our DITA products which provides control over how the output is represented using either of these elements and include a comment which describes the details of the change.

From a comparison perspective grammars which allow singleton choice are harder to deal with, adding a repetition qualifier such as * or + makes representing change easier.

Formatting Element Patterns

The process of marking up mixed content with *formatting* information (by wrapping text in element such as: b, i, span or emphasis) is something which users (document editors or authors) typically do not see or appreciate in XML terms. Explaining that they have deleted some text and replaced it with an element containing some text rarely enlightens the situation. They are generally interested in the words that have changed and as a secondary concern what formatting may have changed. However, they do like to see the added or deleted text represented as it was formatted in the two input files.

To meet this requirement we tend to flatten formatting information. The result is a paragraph or other container of mixed content where all of the words are siblings at the same level of the XML tree. When compared in this re-organized form changes to the text are easier to align and identify. There are a number of flattening techniques that can be used, including using start and end markers, so that for example

`some <b>bold</b> text` becomes
`some <b-start/>bold<b-end/> text` or by moving formatting information to some out-of-band location.

When trying to generate a result we need to reconstruct as much of the original flattened hierarchies around the added and deleted content and in some cases those hierarchies can be in conflict.

We have found two properties of grammars that are very useful when reconstructing a good representation of the input hierarchies in the comparison result which we informally call removability and nestability:

Removability
Can the wrapping element be removed still leaving a valid result? This is true for example, if the elements in the content model of a span or other formatting element, are the same as the content model at the point where the span is used in the containing element.

Nestability
Can the formatting element contain an immediate child of the same type? This is usually true when recursive reuse is used in a content model, for example allowing a span element directly containing another span element.

These properties are often true for established document formats such as DocBook, DITA and (x)html, however for simpler document formats they are not always true and cause difficulty for our algorithms which try to reconstruct the formatting hierarchies. As well as for comparison, we suspect these would also be good properties if the document format were to be supported by a WYSIWIG or authoring editor.

Processing Instructions (PIs) and Comments

Comments and Processing Instructions (PIs) are often used by developers as a simple or ad-hoc extensibility mechanism: “The grammar doesn’t allow me to put this information here.... I know, I’ll write it in a PI and change the reading code to understand those PIs.”

However, it is difficult to describe changes to them because they lack structure. PIs are often used to represent changes using accept/reject or review systems used in editors, and thus are a possible way of describing comparison results. However, when used in this way its almost impossible for accept/reject PI mechanisms to describe changes to PIs. Similarly CSS styling using attributes cannot be used for highlighting comment change.

We would therefore caution their use as an ad-hoc extensibility mechanism. It may work in limited scenarios with writer/reader agreement, but a general purpose comparison has a hard time describing the changes without resorting to similar ad-hoc mechanisms to do so.

Bibliography

[1] Binary codes capable of correcting deletions, insertions, and reversals. Vladimir I. Levenshtein. Soviet Physics Doklady, 1966.

[2] XML Schema Part 2: Datatypes Second Edition. W3C Recommendation. 28 October 2004. http://www.w3.org/TR/xmlschema-2/

[3] International Components for Unicode http://sites.google.com/site/icusite/

[4] Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. 26 November 2008. http://www.w3.org/TR/REC-xml/

Head Office (Sales and Support)

DeltaXML Ltd

Malvern Hills Science Park
Malvern, Worcestershire
WR14 3SZ UK

W: www.deltaxml.com
E: info@deltaxml.com
T: +44 1684 532 130

DELTAXML