Representing Change Tracking in XML Markup

Robin La Fontaine
DeltaXML Ltd
<robin.lafontaine@deltaxml.com>

Tristan Mitchell
DeltaXML Ltd
<tristan.mitchell@deltaxml.com>

Nigel Whitaker
DeltaXML Ltd
<nigel.whitaker@deltaxml.com>

Abstract

This paper presents work done over the past two years to provide an improved change tracking representation for documents in XML. The original intention was to provide improved change tracking for the OpenDocument format (ODF), but the approach is generic and is therefore potentially applicable to other XML document formats and even XML data.

A detailed specification was developed and prototype implementations developed in Abiword and KWord to demonstrate interoperability. However, developers of the main ODF office packages found the approach a challenge and were less keen to implement it and are currently looking at other options.

This paper presents the basic design principles behind the proposal, and how these are satisfied in the approach taken. Since the initial work, there has been interest from the wider XML community and new requirements relating to its use within XML editors have also been proposed. There is now a W3C Community Group formed specifically for change tracking markup, and a standard in this area could have significant benefits for the XML community as a whole.

1. Introduction

The ability to track changes made to text documents is commonly available in document editing systems. These are now moving to XML, e.g. OOXML and ODF [2]. At the same time, structured document formats all use XML and currently do not have any extensive capability to track changes. The change-tracking capability of

XML editors is fairly basic, for example many do not track attribute changes, and there is no common standard. The lack of a standard means that documents with changes tracked cannot be moved between XML editors.

A standard way of tracking changes in XML documents would provide many benefits:

- documents with tracked changes could be moved from one XML editor to another
- XML editors could track changes in any XML document type
- every XML document type could include a change history and the ability to roll back to previous versions
- software designed to handle change in XML could be applied to many different XML document types

The state-of-the-art at present is that every XML document type takes its own approach to change tracking. OOXML is built on the underlying binary model within Microsoft Word. ODF has only a very limited capability to track some changes. DITA uses rev and status attributes to indicate changes and DocBook similarly has a revisionflag attribute, but neither can track attribute changes or complex structural changes.

XML editors track changes either by additional markup or using Processing Instructions (PI). Additional markup has the advantage of being easily processed using standard XML tools but at the cost of modifying the underlying schema. PIs have the advantage of preserving the latest state of the document in valid XML markup but the PIs do not have structure and so are limited in the changes they can track.

The original purpose of the change tracking format described in this paper was to improve the change tracking within ODF. The proposal was known as GCT (Generic Change Tracking) and full details can be found in [1].

ODF is a large and complex XML format, including representation of textual documents, graphics, spreadsheets and presentations. Due to its complexity, it seemed sensible to take a generic approach so that any change to the underlying XML could be represented in a precise and unambiguous way. The scope of the actual change tracking within specific elements of ODF could then be constrained within a modified version of the RelaxNG schema.

It was generally agreed within the ODF community that the generic approach proposed would be applicable to other XML documents. However, the developers of ODF office packages felt that they would not be able to implement such a generic approach because their own internal data structures were very different from the XML representation. They felt that the ability to represent any change was beyond what was needed and too complex for ODF editing applications to handle. Prototype implementations were however completed for two such office packages and these demonstrated both that the proposal could be implemented and that there was interoperability between two independent implementations.

2. Outline of Paper

In this paper we first explore some of the requirements for change tracking within documents, and the basic principles behind the design of the change tracking format.

We outline the reasons behind the two implementation levels for change tracking: the simplest level allows any change to be represented, but in a non-optimal way; the second level allows structural change to be represented, at a cost of more complexity. Issues of validation of changes are discussed.

Changes need to be structured in some way in order to represent interdependency and to provide meta data including timestamps and author details.

Examples are given to show how attribute changes, element deletions and additions, and text changes are represented.

The issues around structural changes are discussed, and the way that these are represented is described.

The paper outlines how the generic ability to track changes can be used independently or integrated into a particular schema, in a way that provides some control over what can be changed.

Finally, the paper outlines some of the as yet unresolved issues.

3. Requirements for XML Change Tracking

3.1. Distinction between 'edit tracking' and 'revision tracking'

Opinions within the ODF group were divided on the subject of the ultimate purpose of change tracking, but the discussion was useful and raised some important issues. Some viewed change tracking as a record of the edits made by an application. Others viewed change tracking as a record of the changes between two revisions of a document. These can be characterized as an 'Application viewpoint' and a 'Document viewpoint', and they lead to two different interpretations of change tracking.

The Application viewpoint asserts that change tracking should support the features in editors, no more and no less. Therefore editor application programmers need to agree on what these edit operations are so that they can be unambiguously represented, in this case in the ODF document format. The Document viewpoint asserts that change tracking should support changes to the document, in this case the XML representation of the document in ODF, so it should be possible to roll back to any previous version.

The Application viewpoint did not want a Document viewpoint solution because it was considered hard to implement. Existing ODF implementations tended to read ODF files into internal editor specific data structures and not keep any association with between the XML representation and the internal one. The Document viewpoint regards the Application viewpoint solution as inadequate for other applications

and a moving target: as editors add new features the standard will need to be changed. It was noted that a Document viewpoint solution must include all of the needs of the Application viewpoint.

This leads to a refinement of the term 'change tracking' to two variants:

'**Edit tracking**' is the ability to record edits made in an editing application in order that they can be viewed, accepted or rejected at a later date.

'**Revision tracking**' is the ability to record changes to a document such that these changes can be displayed in a document viewer, and the document rolled back to a previous version.

When the scope is widened from ODF to any XML, there are many different applications and therefore it would not be possible to have a single standard for edit tracking. If the revision tracking is based on changes to the underlying XML representation, then a single standard is possible and potentially useful across many different XML schemas.

3.2. ODF Requirements

These are the requirements that were discussed for ODF in the Advanced Document Collaboration subcommittee.

1. **Reversibility:** At its most basic level, change tracking, as its name suggests, is the ability to track or record a change to an XML document. The way that a change is recorded needs to be capable of being reversed or undone, typically to support undo and/or redo operations of an editor. Therefore by undoing changes that have been tracked, it is possible to move back to a previous version of a document. A standard for change tracking should include a very clear definition of how a particular change is reversed.
2. **Easy to ignore:** Change tracking information is additional information for a document, and there will always be applications that are not interested in it. Therefore it should be easy to ignore changes that have been tracked, and if the changes are ignored then the result should be the latest version of the document.
3. **Use markup:** Since this is an XML standard with associated schema, it makes sense that the changes should be recorded in standard XML markup.
4. **Granularity:** It is preferable that changes should be tracked at a low level of granularity. For example, it should be possible to represent the deletion of a single word or a single character within a larger PCDATA segment and not have to delete all of the textual content, and then add new text without the word or character.
5. **Grouping:** Although some changes are quite simple, other changes, such as a global search and replace operation or the deletion of a column in a table, are more complex although they still need to be considered as a single reversable

change operation. This implies that the ability to group simple changes together into more complex units is necessary.

6. **Dependency:** As mentioned above, it is fundamental that the tracked change can be reversed. The reversal of changes does however introduce a new issue, which is that it is not always possible to reverse one change without first reversing, or accepting, some other change. Consider for example the case of a spelling correction to some text inside a paragraph which is them deleted entirely.
7. **Deleted content outside document body:** In an ODF document, all of the text within the main body of the document is deemed to be a part of the document. The advantage of this is that an application that does not understand particular markup can ignore it and just process the content. This means that any deleted item will be represented by a marker, and the actual deleted content will be elsewhere in the document. This requirement was to maintain compatibility with previous versions of ODF.
8. **Change tracking markup integrated with schema:** The markup for change tracking needed to be part of the schema (RelaxNG) for ODF.

3.3. General XML Requirements

Most of the above requirements were established in the initial work with ODF. When looking more generally at requirements for XML editors and authoring tools some of the requirements may be adjusted from those for ODF. Note that these reflect our view which does not necessarily reflect the views of others involved in the W3C community group.

The scope of a change tracking standard also needs to be considered. For example, it would probably be sensible to exclude the representation of changes to a DTD, and probably also exclude changes to entities. Changes to processing instructions and comments would ideally be included, but they are likely to introduce an extra degree of complexity.

A common practice for XML editors is to use Processing Instructions (PIs) to represent changes, and therefore it seems highly desirable to have a representation that uses processing instructions. The reason that XML editors use processing instructions is so that the document itself remains valid relative to its schema. The ability to swap between a markup representation and a processing instruction representation, in a way that is completely lossless, would make a change tracking standard more versatile.

A processing instruction representation was not provided in the ODF work but a simple approach to this would be to convert the outermost element into a processing instruction, and its content becomes the processing instruction content. This approach needs to be validated. Initial experimentation using the oXygen editor suggests that it would work subject to some constraints.

Another implication of the use of processing instructions is that the deleted content can be represented in situ, and does not need to be moved to another part of the document. Therefore the requirement noted above that deleted content should be outside the document body is not necessarily a requirement for uses in other XML document formats.

3.4. Validation of changes

It is certainly necessary to define whether or not a particular change is valid. This is potentially very complex. However, XML provides many mechanisms for validating an XML document, whether it is against a schema or with additional Schematron rules or NVDL validation. In order to validate changes, the validation must take account of the document that is being changed.

We can circumvent this complex issue by saying that if the document before the change is valid, and if the document after the change is valid, then the change is valid. This is simple and intuitive to understand, and completely removes the need for more complex validation of changes.

4. Levels of Complexity

It is always good to strive for the simplest possible solution to a problem. But simple solutions can be limited, and increasing complexity is sometimes needed to provide a more useful solution.

It is possible to represent any change to an XML document using just the deletion and addition of elements. However, this is not particularly useful, because it would mean that an entire subtree would need to be deleted and added in order to represent the change to a single attribute. Similarly, it is evident that the ability to represent changes to textual content is necessary. This leads to a basic level (Level 1) of change tracking ability which would include the addition and deletion of elements, the addition and deletion of text, and the addition, deletion and modification of attributes.

As an aside, it may seem odd that only attributes can be modified, and not text or elements. It turns out to be convenient to have a single operation to modify the value of an attribute, whereas there is little to be gained by having a special operation for modification of text: modify-text has no advantage over deletion and addition. Regarding elements, modification of an element involves some change to its attributes or content, so there is no need for a special modify-element operation.

Although this basic level of complexity seems to be adequate at first sight, it soon becomes clear that it is not always sufficient. This is certainly the case when the XML is used to represent documents. This is simply because XML documents tend to use structural changes to XML in order to represent changes that a viewer might consider to be only aesthetic, for example the addition of text decoration.

Another example of this is when an editor inserts a newline in the middle of a paragraph, i.e. splits it into two paragraphs. Level one represents this as a paragraph with deleted content and another added paragraph, however an editor does not expect to see change to those paragraphs, but rather the insertion of a new line. Such changes do not always fit well with the underlying XML structure.

In order to avoid the need to delete and add potentially large amounts of content in order to show such changes, we need to introduce the ability to add and delete structural information, i.e. XML tags. This does make the change tracking much more precise, though at a cost of additional complexity. It may also be argued that a typical XML editor will allow the addition of structural markup inserted around existing content, so again there is a need for a more precise representation of this.

Therefore, **Level 1** provides the ability to modify attributes, add and delete elements, and add and delete text. It also enables changes to be grouped into transactions where a single transaction moves the document from one valid state to another.

Level 2 adds to this the ability to add or delete element structure around existing content and to split and merge elements in more complex ways.

In the following sections we will describe these two levels, and show examples of how changes are represented. They are not intended to be a formal definition of how the change tracking format works, but rather an introduction.

5. Level 1

5.1. Change Transaction (CT) Structure

This structure provides a place for metadata and a structure to define any dependencies between changes.

There must be a position in the document where the change transactions are defined, each being identified by an ID. Each will have some associated meta information such as the name of the author who made the change, and the time and date.

The ordering of the change transactions is important. If a user wishes to undo the changes one by one, then this can be achieved by undoing the last change transaction, i.e. it is a stack of transactions.

It is also possible to group CTs in a change transaction group (CT group). This will have similar meta information to a CT. All the members must be previously-defined CT or CT groups. The effect of undoing a CT group will be to undo a number of CTs.

A CT group may be ordered (a CT stack) or unordered (a CT set). The members of a CT set can be accepted or rejected in any order. The members of a CT stack must be accepted or rejected in the defined order, i.e. undo last member first. An example of a CT set would be a global textual replace operation, the user may wish to accept all of the replacements as one operation, or accept/reject them individually

in any order. A change that depends on another change would be represented together as a stack.

5.2. Changes to Attributes

Attribute changes are tracked within new attributes. The reason for doing it this way is to make the minimum structural changes to the document. Typically, there will only be one or two attribute changes within an element, although if there were a large number of changes then this would not be very readable.

An element will always contain the latest version of its attributes. This means that if an attribute is added, we only need to record the fact that it has been added, and not its value because the value will be specified in the element. When we change the value of an attribute, then we need to keep a record of the previous value so that the change can be undone. Similarly, for attribute deletion we need to keep a record of the original value.

Each attribute change will reference a change transaction, and this and other information is encoded in a new attribute which is in a defined namespace, but the actual name of the attribute is generated. The information that is encoded within the attribute value is as follows:

1. The change transaction (CT) ID. This is a reference to the ID.
2. The type of change: insert, remove, modify
3. The name of the attribute that is changed
4. The old value of the attribute – this is not needed for an added attribute because the value will either be in the element or, if the attribute is later deleted it will be recorded there.

Example 1. Attribute addition: the outline-level attribute has been added

```
<text:p text:style-name="Standard" text:outline-level="3"
   ac:change001="ct1,insert,text:outline-level">
   How an attribute is added
</text:p>
```

Example 2. Attribute deletion: the outline-level attribute has been deleted

```
<text:p text:style-name="Standard"
  ac:change001="ct1,remove,text:outline-level,3">
  How an attribute is deleted
</text:p>
```

Example 3. Attribute modification

```
<text:p text:style-name="Code"
  ac:change001="ct1,modify,text:style-name,Standard”>
```

```
  The style on the paragraph has been changed from Standard to Code
</text:p>
```

5.3. Changes to Elements

An element is marked as inserted with an attribute, delta:insertion-type="insert-with-content" (we will discuss the other insertion types later). There will also be a reference to a CT which will have all the meta-data associated with this change.

Example 4. Element insertion

```
<text:p delta:insertion-type="insert-with-content"
        delta:insertion-change-idref='ct1234'>
        This paragraph is inserted.
</text:p>
```

When an element or other content is deleted, it is wrapped in an element <delta:removed-content/> to indicate it is no longer part of the document. This element can contain mixed content, ie one or more elements and/or text that was removed as part of the same editing operation.

Example 5. Element deletion

```
<delta:removed-content delta:removal-change-idref='ct456'>
                <text:p>    This paragraph is deleted.   </text:p>
</delta:removed-content>
```

Note that a deleted item may contain changes within it, but the changes must all be before its deletion.

5.4. Changes to Text

Text addition uses the conventional method of setting a marker at the beginning of the addition and a corresponding marker at the end. These markers are empty elements, and they are linked using an ID.

Example 6. Simple text insertion

```
<text:p>
How text is
   <delta:inserted-text-start delta:inserted-text-id="it632507360"
     delta:insertion-change-idref=”ct1”/>
very easily
   <delta:inserted-text-end delta:inserted-text-idref="it632507360"/>
added.
</text:p>
```

Additions may not always be within a single element, but the delta:inserted-text-start and delta:inserted-text-end must both have the same parent element when they are created, and the content between them must be PCDATA only. Therefore when a second paragraph is added as per the example below, the first atomic change terminates and the paragraph is added in the normal way. The CT reference provides a link to indicate these occur at the same time as a single addition. This avoids having two ways to add an element and avoids the need to track across the element hierarchy to find the corresponding end of an addition.

Example 7. Text insertion that flows into a new paragraph

```
 <text:p>
How text is
   <delta:inserted-text-start delta:inserted-text-id="it123"
      delta:insertion-change-idref=”ct3”/>
very easily added.
   <delta:inserted-text-end delta:inserted-text-idref="it123"/>
</text:p>
<text:p delta:insertion-type="insert-with-content"
      delta:insertion-change-idref=”ct3”>
And the addition is into a second paragraph.
</text:p>
```

Additions must therefore always be non-overlapping and the start and end of a change must be within a single element, when they are formed. Of course they may not be within a single element at some later stage due to other changes, but in this case it would not be possible to 'undo' it. This rule adds clarity at the slight cost to the writer application, i.e. the application creating the change, and the considerable gain for the reader application, i.e. the application consuming the change. Since any number of atomic changes can be associated with a single CT, there is no loss of information.

Text is marked out as deleted in exactly the same way as an element is marked as deleted, i.e. it is wrapped within a change tracking element.

Example 8. Simple text deletion

```
 <text:p>
How text is
  <delta:removed-content delta:removal-change-idref=”ct2”>
   deleted or </delta:removed-content>
removed from a paragraph.
</text:p>
```

If the content is not simple text, but mixed content, it is handled in the same way.

Example 9. Mixed content deletion

```
 <text:p>
How text is deleted
   <delta:removed-content delta:removal-change-idref=”ct2”>
     or <text:span text:style=”bold”>removed</text:span> like this
   </delta:removed-content>
from a paragraph.
</text:p>
```

6. Level 2

6.1. Add an element around some existing content (insert-around-content)

In document editing, it is common to add text decoration or structural information to existing content. As the content itself is not changed, we wish to reflect just the addition or change of the structure.

Example 10. Addition of a element around some text

```
 <text:p> This text will be made
   <text:span text:style-name="bold-style"
     delta:insertion-type='insert-around-content'
     delta:insertion-change-idref='ct1234'>
   bold
   </text:span>
  . </text:p>
```

Since this tag has been added around the content, when the change is undone its content will remain in place.

6.2. Delete an element but not its content (remove-leaving-content)

This is the opposite of the previous example, where the tags around an element are removed but the content remains.

Example 11. Removal of a element leaving the text

```
<text:p> This text will be made
  <delta:remove-leaving-content-start delta:removal-change-idref='ct345'
     delta:end-element-idref='ee888'>
   <text:span text:style-name="bold-style" />
  </delta:remove-leaving-content-start>
unbold
  <delta:remove-leaving-content-end delta:end-element-id='ee888'/>
. </text:p>
```

Since the element has been deleted, but the content remains, it is split it into a start and end element so that the content remains in position at the correct level. The split element is linked by an ID so that it can be reconstructed. The splitting of a wrapper element into its start element and end element means that deleted wrapper elements do not contribute to the hierarchical structure of the document. This is important because over time they may be split across element boundaries. When created, the start and end elements must have the same parent element.

6.3. Split an element into two elements (split)

The classic example of this is when a paragraph is split into two by the insertion of a new line. Similarly, a list item might be split into two list items. The element that is split is known as the parent of the split and the element that is created is known as the child of the split.

Example 12. Two text:p elements formed from splitting a single text:p element

```
<text:p split:split01='sp1'>
This paragraph will be split into two.
</text:p>
<text:p delta:split-id='sp1'
  delta:insertion-type='split' delta:insertion-change-idref='ct1' >
This will be in the second paragraph.
</text:p>
```

This facility allows the representation of quite a common editorial action. Note that there may be elements between the split paragraph elements but these would all have been added in the same or a later CT. Therefore there is an attribute value pair to link the start and end of a split.

6.4. Merge two sibling elements into one (merge)

This merge change is the opposite to a split, and is used to capture a number of common editing operations, for example moving the cursor to the start of a paragraph and pressing the delete key to join it to the previous paragraph. This would create a single paragraph as shown in Example 13

Example 13. A single text:p element formed from merging two text:p elements

```
<text:p text:style-name="Standard">
These paragraphs will be merged into one.
 <delta:merge delta:removal-change-idref='ct2'>
   <delta:leading-partial-content/>
   <delta:intermediate-content/>
   <delta:trailing-partial-content>
```

```
      <text:p text:style-name="Code"/>
    </delta:trailing-partial-content>
  </delta:merge>
This was in the second paragraph.
</text:p>
```

The delta:leading-partial-content element is in this case empty, but it could contain content from the first text:p element. Similarly, the delta:intermediate-content element is also empty, but it could contain any number of elements or content that lay between the two text:p elements. The delta:trailing-partial-content always contains one element, i.e. the element that forms the end of the merge operation. This element may or may not contain content. Any content within it would have been removed from the start of the final element in the merge. The element in delta:trailing-partial-content could be of a different type to the one that encloses the delta:merge; this allows elements of different types to be merged.

The above is only a simple example, but the structure allows for more complex merge operations to be represented and to be reversible. Such a use-case is depicted in Figure 1 which is a screenshot made during the editing of this paper. The region highlighted in blue has been selected (by holding and dragging the mouse). Pressing the delete key would logically complete the merge operation and the highlighted text would be replaced by a merge element containing the three children with the content indicated.

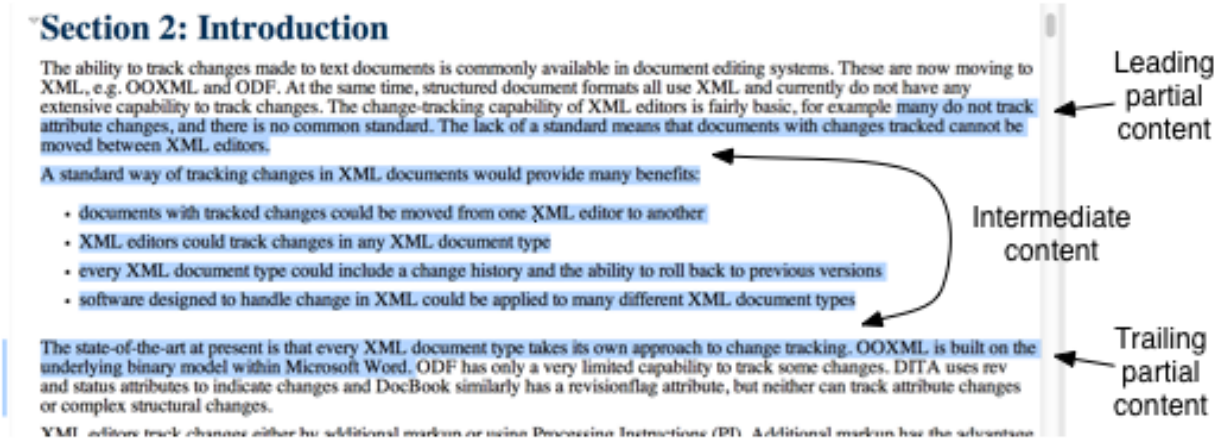

Figure 1. Merge regions

The merge operation could be represented using remove-leaving-content and insert-around-content but this leads to a more complex structure. Therefore the merge element provides a special representation for this common editing action.

7. Integration of Change Tracking with a Host Schema

7.1. Stand-alone use

The format can be used as an independent addition to an existing XML host format. In this scenario no changes are made to the schema of the host format, but the track change elements and attributes are used to represent changes and edits to a document.

This is the simplest way to use change tracking, but of course the document cannot be validated against a schema. However, the latest version of the document can easily be extracted and checked, and each change can be individually rolled back and the resulting document checked. This could be handled using NVDL validation.

A Schematron checker can also validate the entire document with change tracking against certain rules, for example that references to change transactions are correct.

7.2. Schema integrated use

In this scenario there will be a RelaxNG schema which specifies the host format with change tracking schema integrated with it. The stand-alone testing mentioned above would still be valid and work, but as well as that the change-tracked document could be checked against a schema.

The steps outlined below show how to change a schema to allow changes to be tracked throughout the schema. By restricting the application of each individual step, it would be possible to restrict the tracking of changes to certain areas within the schema.

Integration of Level 1 is simpler than integration of Level 2.

Schema Integration Level 1

The following steps provide a way to perform the integration.

Step 1: An element containing the change meta data (change transactions and their grouping) must be allowed at one point in the document.

Step 2: Any element in the host format that has one or more attributes which can be added, deleted or values changed, need to allow attributes in the ac: namespace.

Step 3: All elements that can be added or deleted with their content (including any element that allows no content, i.e. is always empty) need to allow the attribute delta:insertion-type with value 'insert-with-content' and be permitted as a child of delta:removed-content. Note that this is not necessarily all elements, for example an element that is only used as a required item and never in a choice would not be

in this category. Some modification to choice element structure may be needed to allow changes.

Step 4: All elements that allow element content must have their content model modified so that they allow delta:removed-content to appear anywhere as a child element.

Step 5: All elements that allow PCDATA content, including elements that allow mixed content, need to allow for PCDATA content to be added (Step 4 allows text to be deleted).

Schema Integration Level 2

Step 1, 2, 3 and 5 are as Level 1. Step 4 is replaced with Step 8.

Step 6: Any element that can be added as a wrapper around existing content, or removed as a wrapper (in this situation it is often true that the content model of the element is a subset of the content model of its parent): These elements need to allow delta:insertion-type='insert-around-content' and supporting attributes.

Step 7: For any Step 6 element, if any content is required then the content model must be changed to make an empty element (no content) allowed when its parent is delta:remove-leaving-content-start.

Step 8: This is an extension to Step 4: Any element that allows content must allow as child elements delta:remove-leaving-content-start, delta:remove-leaving-content-end, delta:removed-content and delta:merge to appear zero or more times anywhere.

Step 9: Elements where it is useful to represent a split or merge. Typically these will have mixed content, though this is not a condition. These elements need to allow delta:insertion-type='split' (and supporting attributes).

8. Conclusions

The description provided in this paper gives a flavour of how change tracking could be achieved generically for any XML format. Initial implementation and validation of this has been performed for the ODF format, but it has not been tested on other XML formats. The processing instruction representation has also not been worked through in detail, nor implemented.

One area of change tracking that has not yet been considered is the representation of conflicting changes. These might result from real-time collaboration or the merging of concurrent edits.

This initial work will be submitted to the W3C community group for their consideration and potential development into a recommendation. As outlined above, there are potentially significant gains for the XML community in having a robust and generic standard in this area, and it is hoped that the new community group will be able to achieve this.

Bibliography

[1] Robin La Fontaine: XML Change Tracking ODF Proposal submission http://www.deltaxml.com/attachment/481-dxml/XML-change-tracking.pdf

[2] Open Document Format for Office Applications (OpenDocument) Version 1.2 29 September 2011, OASIS Standard. http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.pdf