A Generalized Grammar for Three-way XML Synchronization

Robin DeltaXML Ltd [http://www.deltaxml.com] La Fontaine, DeltaXML Ltd [http://www.deltaxml.com]

Nigel DeltaXML Ltd [http://www.deltaxml.com] Whitaker, DeltaXML Ltd [http://www.deltaxml.com]

Abstract

Synchronization problems, though diverse, can be categorized into distinct classes. For example, collaborative document editing often involves complex merging of multiple XML documents, with rule sets such as "mark conflicting edits for manual attention but accept all other edits" - a different rule set would typically be used in translation, synchronizing Original English, Modified English and Original Japanese texts. This processing can be described as a controlled merge of three data sets. In practice the execution of the merge, while falling into clear categories, is different in each case, leading to the need to express and formalize rule sets.

A naive approach describes a specific synchronization algorithm in terms of additions, deletions and conflict identification. However, such a fixed algorithm is inflexible and meets only limited needs, typically providing a solution for only one problem space. A more flexible method for describing and specifying how the synchronization is to be performed is required to allow us to categorize and solve sets of related problems.

This paper proposes a general synchronization grammar which can describe synchronization rule sets. For example, when handling three input files, we show that changes to elements can be described in terms of just seven possible permutations. Similarly, PCDATA and attribute changes can be described in terms of a fixed set of permutations. Using these permutations a grammar is proposed, allowing precise description of synchronization algorithms and rule sets and providing a testable framework for their implementation.

The paper applies the resulting grammar to existing synchronization tools and technologies and shows how the grammar can be applied to provide solutions for specific application areas, including document workflow and translation.

The paper will be of particular interest to architects, to project managers and to programmers working with complex documents and with workflows involving multiple concurrent changes.

Table of Contents

Introduction

Three-way XML synchronization is concerned with the intelligent merge of three data sets represented in XML in order to generate a result data set formed from selected parts of the original three.

Two and three way merge is described in an earlier paper [SYNC]. The approach described there was essentially an implementation of one set of rules for generating a result data set from the three input data sets, on the basis that one of the sets represented a 'base' and the two others were derivatives of this. By detecting the additions, deletions and modifications between the base and the two derivatives, the edit actions could be combined to generate a result. Where possible, the result combined the edits from both paths. Where conflicts occurred, these were identified and reported.

Further analysis suggests that this was but one example and there are a number of similar synchronization problems. Although diverse, these can be categorized into distinct classes. This paper proposes a general synchronization grammar which can describe synchronization rule sets.

In order to illustrate this, we will first describe two of these problem spaces and then see how they can be seen as the same problem with different rule sets for generating a merged result document. The two examples are the resolution of concurrent edits made to a single document and the update of a translated document. Other examples, not described here, include the update of a configuration file which has been customized and the original is then updated: the problem is to merge the customization changes with the changes made to the updated version. A similar problem exists for any generated XML data which is then hand-edited: when the generated XML is updated the manual edits need to be merged back in to this new base.

For each of the two examples, we first provide informal rules to describe what we are trying to achieve. We then establish the full set of possible element combinations that are availabe in the context of three original data files. A grammar is then proposed for specifying the rules as to how a result is achieved from each possible combination. The original informal rule sets for each example is then re-visited and the formal rule set defined.

The proposed grammar has a number of benefits. It provides a more precise description of a synchronization requirement or algorithm. It also forms the basis for a flexible, data-driven implementation.

In order to keep the paper to a reasonable length, we consider only the possible rules and combinations for elements. There is a corresponding set of combinations for PCDATA and attributes which can be handled in an analogous manner.

Nomenclature

We are considering three XML documents. We will identify these generically as A, B and C. It is important to distinguish between these because in the rule sets we are describing these are not interchangeable. The result of our synchronization operation is the document denoted D.

Informal Examples

Two Concurrent Edits of One Document

In this scenario, we can consider A, B and C as three documents as follows: A is a base file and B and C are two derivatives, each edited in a different way. Our task is then to generate a synchronized document where any common edits are propagated, because there is no conflict. This would typically be defined loosely in a rule set as follows:

Example 1. Concurrent Edit Rules

1. "If an element is unchanged, it appears in the result."
2. "If an element is deleted in B or C, it does not appear in the result."
3. "If an element is changed in either B or C, the changed element appears in the result."
4. "If an element is changed in both B and C, this is a conflict and we select the changed element C to appear in the result."
5. "If an element is not in A but is in one of B or C, the new element appears in the result."
6. "If an element is not in A but is in both B and C, and is the same in both, the new element appears in the result."
7. "If an element is not in A but is in both B and C, and is different in them, this is a conflict and we select the new element from C to appear in the result."

Update of a Translated Document

In this scenario, we have an original English document and a translation into Japanese, and an updated English document. These relate as follows: A is the base file, i.e. the original English, B is the translation of this file, with the same structure but different text, and C is the modified English document.

Our task is then to generate a synchronized document which will be a template for the revised translated document. This means that only those parts of the document that have been changed need to be re-translated. In this result file, any unchanged English is represented as the unchanged translation. For changed text we need the details of the changes in the result file so that the translation can be manually updated. This would typically be defined loosely in a rule set as follows:

Example 2. Translation Rules

1. "If an element is unchanged, it appears in the result."
2. "If an element is deleted in C, it does not appear in the result."
3. "If an element is changed in C, the elements from A, B and C appear in the result."
4. "If an element is the same in A and C, then B appears in the result."
5. "If an element is added in C, it appears in the result."
6. "If an element is added in B, it appears in the result (because it was 'special' to the translation)"

This translation problem is more fully described in [BABEL]. This introduces the concept of a 'unit of translation' which is an element that we treat as a whole and do not sub-divide. For example, a paragraph is a unit of translation and we do not track changes below this level - this is simply because

trying to automate word-by-word translation is unlikely to result in anything useful. A translator would rather work at the paragraph level.

Combinations and Actions

Element Combinations

What Combinations of Elements are Possible?

We can see that a pattern is emerging as we look at the different applications and how they resolve the different situations. The first step to formalizing this is to enumerate all the possible situations. Then we can review how these can be handled.

Given three initial data sets, A, B and C, we have the following possible combinations to describe whether an element is present in each of the data sets:

- Only one present: [A], [B], [C]
- Only two present: [A, B], [A, C], [B, C]
- All three present: [A, B, C]

This is a total of seven combinations. But we need to go one step further because the informal rules above also consider whether, for a given combination, the data is the same in all the files or not. In other words, if an element is present we need to know if it is unchanged or changed in the different data sets. This clearly does not apply if there is only one element present, so there are only another seven combinations. This gives us a combination set as follows:

1. [A]
2. [B]
3. [C]
4. [A, B] A == B
5. [A, B] A != B
6. [A, C] A == C
7. [A, C] A != C
8. [B, C] B == C
9. [B, C] C != C
10. [A, B, C] A == B == C - all equal
11. [A, B, C] A != B != C - all different
12. [A, B, C] A == B, A != C - c is different
13. [A, B, C] A == C, A != B - b is different
14. [A, B, C] B == C, A != B - a is different

When we say, for example, we have an element with a 'version set' of [A, B] and that A == B, we mean that the subtree corresponding to the A input is (deep)equal to the subtree corresponding to the B input. Another way of looking at this is that the subtree corresponding to this element and all nodes underneath it also have a version set of [A, B].

Axioms

1. The root of the tree must contain [A, B, C] - this implies all three inputs have the same root element starting point.

2. For each of the children of any parent element in the tree the version set of a child element is a subset of the version set of the parent element - we cannot include a child if we have not included its parent - it is not possible for an element to be 'parent-less'.

3. If all items in a subtree have the same version set, then it is unchanged.

Actions: Selection and Recursion

We have introduced a set of 'patterns' or 'matches' which describes all possible combinations of inputs. Each node in the input tree must correspond to one of the patterns above. However, we need to define how the synchronization result is produced. Our technique is to define a number of actions that can be performed when each of the patterns is matched. At this point we will also introduce fragments of our syntax used to represent the patterns described above.

For the first three simple patterns, [A], [B] or [C], the actions which can be performed are very simple. It is possible to either include or exclude the appropriate subtree in the result at that point. More generally, it is also possible to include the current element we are considering and then recurse and apply rules to all the child nodes (elements, attributes etc.) at this point. In this case ([A], [B] or [C]), as there is only a simple single-input subtree it may seem that there is little difference between recursion and subtree inclusion. However, other examples will demonstrate more obvious uses of recursion and we will also see how recursion will be useful when controlling the 'granularity' of synchronization.

For the pattern [A], [B] or [C], given that our version set corresponds to a single input tree, there is only one subtree which can be included when one of these three patterns match. Our pattern and action grammar will therefore follow the following form, described here for pattern [A]:

```
<versionset-a>
  <a-subtree include="yes"/>
  <include-and-recurse include="no"/>
</versionset-a>
```

The `a-subtree` element indicates whether or not to include the subtree from the A data set, according to the value of the `include` attribute. The `include-and-recurse` element indicates whether or not we are to include this element and then recurse through its children.

We can now consider the six patterns with two input version sets. There are three patterns where the descendent subtrees are not equal, for example for an element with versionset [A, B] we also have A != B (subtree inequality). In this case we could include the subtree corresponding to the A input, and/or that corresponding to the B input. Our final option is to include the currently matched element and apply rules to the child nodes. An example of the patterns is shown below:

```
<versionset-ab-a-ne-b>
  <a-subtree include="yes"/>
  <b-subtree include="no"/>
  <include-and-recurse include="no"/>
</versionset-ab-a-ne-b>
```

There are three other patterns for the two element version sets, the cases with subtree equality. In these cases, choosing the a-subtree is no different from the b-subtree (because they are equal) and so we merge these possible actions into one:

```
<versionset-ab-a-eq-b>
  <ab-subtree include="yes"/>
```

```
  <include-and-recurse include="no"/>
</versionset-ab-a-ne-b>
```

For the three element version set there are five combinations of subtree equality/inequality we need to consider. The form of these follows those introduced previously with actions for subtree inclusion and recursion. To save space we have not included examples here, instead please consult the DTD or complete examples included in the section called “Rule Set DTD”.

Note that the above list is not mutually exclusive, we can conceive of cases where including both subtrees is a possibility, and perhaps they would both appear in sequence in the result. In this case it may be useful to distinguish the two included subtrees, perhaps using comments or 'wrapper elements'. These possibilities will be discussed later, in the section called “Synchronization Annotations”.

Recursive actions

Consider the following example of element content:

A: <x><a/></x>
B: <x><a/><b/></x>
C: <x><a/><c/></x>

The `a` child element is consistent through all three inputs (and therefore has versionset [A, B, C]). However, in two of the inputs something 'different' has happened. In some synchronization scenarios this could be regarded as a conflict - the second child of `x` is different. However, consider the case where the b and c are for example Docbook elements. One author may have tried to expand/clarify a topic by adding a diagram, the other by adding a paragraph of explanatory text. In this scenario these are two distinct additions, both equally valid; it seems sensible that the result should be: `<x><a/><b/><c/></x>`.

If we say the addition of distinct elements is not a conflict, is it possible to have an element conflict? Perhaps, if the same element is added but with different content:

A: <x><a/></x>
B: <x><a/><y><y1/></y></x>
C: <x><a/><y><y2/></y></x>

Here we could say two conflicting `y` elements have been added. But we decided not to take this approach. The element `y` is common to both B and C input trees, and therefore has a version set of [B, C]. In the merged or unified tree the `y` element has two children: a `y1` with versionset [B] and a `y2` with versionset [C]. We can recurse into the children and apply rules to process them at that level of the tree. The same is true even if element `y` contained PCDATA, although it is more complex to represent this because we need to wrap the two distinct PCDATA strings.

Recursion is often the preferred option and should be applied where possible/appropriate in order to report changes and/or conflicts at the finest granularity. In an extreme case, given the first Axiom, it would be generally unhelpful to present a number of subtrees at the root level when the actual differences appear several levels further down.

Elements, Attributes and Text

So far we have only provided examples of elements, but XML is slightly more complex! While we believe that the patterns we have introduced can be used to describe other types of tree nodes such as attributes or text, the actions required will be slightly different.

Consider the following example of an element with some attributes:

A: <x a='2'/>
B: <x a='3' b='42'/>
C: <x a='4' b='hello'/>

In our merged or unified representation of the three inputs we follow the XML model and have a single attribute node in our tree. However, unlike the InfoSet/XPath tree model, we have the concept of attribute value nodes as children of attribute nodes:

```
  Node            Type            VersionSet

x-+               element         [A, B, C]
  +--a--+         attribute       [A, B, C]
  |     +--2      attributevalue  [A]
  |     +--3      attributevalue  [B]
  |     +--4      attributevalue  [C]
  |
  +--b--+         attribute       [B, C]
        +--42     attributevalue  [B]
        +--hello  attributevalue  [C]
```

In this example, the attributes are in conflict between the different input trees, but unlike the example with elements (the section called “Recursive actions”) we cannot add multiple `a` or `b` attributes to the result and still have valid XML. For this reason, we cannot apply our usual rules recursively: we need to know about the set of attribute value nodes in order to identify a conflict. For example, in the case of attribute `a` above, the presence of distinct attribute value nodes implies a conflict which we cannot deal with using recursion. We must decide at this point which value or values to output in the result tree.

This difference means that for processing attributes we need to consider both the leaf, or attribute value, version sets, together with the version set of the immediate parent node in the tree. Lack of space prevents expanding the rules in the paper, but they follow a similar style to those for elements.

The handling of PCDATA strings is analogous to handling attribute values, and again is not detailed in the paper.

Rule Set Definition

We are now in a position to define the rules above in a more formal manner, and apply this to the informal examples presented above. The rule set is presented as a DTD, being the most compact schema representation.

Rule Set DTD

The DTD introduces `wrapper-element` used for annotation which is discussed in the section called “Synchronization Annotations”.

Example 3. Rule Set DTD

```
<!ELEMENT a-subtree (wrapper-element?)>
<!ATTLIST a-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT b-subtree (wrapper-element?)>
<!ATTLIST b-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT c-subtree (wrapper-element?)>
<!ATTLIST c-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT ab-subtree (wrapper-element?)>
<!ATTLIST ab-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT bc-subtree (wrapper-element?)>
<!ATTLIST bc-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT ac-subtree (wrapper-element?)>
<!ATTLIST ac-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT abc-subtree (wrapper-element?)>
<!ATTLIST abc-subtree include (yes|no|true|false) #REQUIRED >
<!ELEMENT include-and-recurse (wrapper-element?)>
<!ATTLIST include-and-recurse include (yes|no|true|false) #REQUIRED >

<!ELEMENT versionset-a (a-subtree, include-and-recurse)>
<!ELEMENT versionset-b (b-subtree, include-and-recurse)>
<!ELEMENT versionset-c (c-subtree, include-and-recurse)>
<!ELEMENT versionset-ab-a-eq-b
      (ab-subtree, include-and-recurse)>
<!ELEMENT versionset-ab-a-ne-b
      (a-subtree, b-subtree, include-and-recurse)>
<!ELEMENT versionset-bc-b-eq-c
      (bc-subtree, include-and-recurse)>
<!ELEMENT versionset-bc-b-ne-c
      (b-subtree, c-subtree, include-and-recurse)>
<!ELEMENT versionset-ac-a-eq-c
      (ac-subtree, include-and-recurse)>
<!ELEMENT versionset-ac-a-ne-c
      (a-subtree, c-subtree, include-and-recurse)>
<!ELEMENT versionset-abc-a-eq-b-eq-c
      (abc-subtree, include-and-recurse)>
<!ELEMENT versionset-abc-a-ne-b-ne-c
      (a-subtree, b-subtree, c-subtree, include-and-recurse)>
<!ELEMENT versionset-abc-a-eq-b
      (ab-subtree, c-subtree, include-and-recurse)>
<!ELEMENT versionset-abc-a-eq-c
      (ac-subtree, b-subtree, include-and-recurse)>
<!ELEMENT versionset-abc-b-eq-c
      (a-subtree, bc-subtree, include-and-recurse)>
<!ELEMENT element-set (versionset-a, versionset-b, versionset-c,
          versionset-ab-a-eq-b, versionset-ab-a-ne-b,
          versionset-bc-b-eq-c, versionset-bc-b-ne-c,
          versionset-ac-a-eq-c, versionset-ac-a-ne-c,
          versionset-abc-a-eq-b-eq-c,
          versionset-abc-a-ne-b-ne-c,
          versionset-abc-a-eq-b,
          versionset-abc-a-eq-c,
          versionset-abc-b-eq-c)>
<!ATTLIST element-set match CDATA #REQUIRED>
<!ELEMENT rule-set (element-set)*>
<!ELEMENT wrapper-element EMPTY>
<!ATTLIST wrapper-element namespace CDATA #IMPLIED>
<!ATTLIST wrapper-element localname CDATA #REQUIRED>
```

Rule set for Two Concurrent Edits of One Document

In the formal rule set shown below, we decide at each point which element to include in the result. Where there is a possible conflict, we output the problem element wrapped in a new element to annotate the warning.

Example 4. Combination Grammar for Concurrent Edits

```
<rule-set>
  <element-set match="*">>
    <versionset-a>
      <a-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-a>
    <versionset-b>
      <b-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-b>
    <versionset-c>
      <c-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-c>>
    <versionset-ab-a-eq-b>
      <ab-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ab-a-eq-b>
    <versionset-ab-a-ne-b>
      <a-subtree include="no"/>
      <b-subtree include="yes">
        <wrapper-element namespace="dxwarn"
          localname="modifiedEdit1DeletedEdit2"/>
      </b-subtree>
      <include-and-recurse include="no"/>
    </versionset-ab-a-ne-b>
    <versionset-bc-b-eq-c>
      <bc-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-bc-b-eq-c>
    <versionset-bc-b-ne-c>
      <b-subtree include="yes">
        <wrapper-element namespace="dxwarn"
          localname="addedEdit1AddedEdit2"/>
      </b-subtree>
      <c-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-bc-b-ne-c>
    <versionset-ac-a-eq-c>
      <ac-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ac-a-eq-c>
    <versionset-ac-a-ne-c>
      <a-subtree include="no"/>
      <c-subtree include="no">
        <wrapper-element namespace="dxwarn"
          localname="modifiedEdit2DeletedEdit1"/>
      </c-subtree>
      <include-and-recurse include="no"/>
    </versionset-ac-a-ne-c>
    <versionset-abc-a-eq-b-eq-c>
      <abc-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-b-eq-c>
    <versionset-abc-a-ne-b-ne-c>
```

```
      <a-subtree include="no"/>
      <b-subtree include="no"/>
      <c-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-abc-a-ne-b-ne-c>
    <versionset-abc-a-eq-b>
      <ab-subtree include="no"/>
      <c-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-b>
    <versionset-abc-a-eq-c>
      <ac-subtree include="no"/>
      <b-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-c>
    <versionset-abc-b-eq-c>
      <a-subtree include="no"/>
      <bc-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-b-eq-c>
  </element-set>
</rule-set>
```

Rule Set for Update of a Translated Document

In developing the formal rule set for this example, we need to consider separately the elements that form a unit of translation and other elements. In the example below, we consider the `para` element as being the only unit of translation, and we establish how to deal with it. Notice that there is no recursion for the `para` element, we deal with each occurrence of this element by selecting whole subtrees and we never go down to a lower level.

When we write the rules for other elements, we tend to recurse to lower levels where we cannot make a decision at this level. There are some cases where we can make a decision, for example:

```
<versionset-abc-a-eq-c>
  <ac-subtree include="no"/>
  <b-subtree include="yes"/>
  <include-and-recurse include="no"/>
</versionset-abc-a-eq-c>
```

This is the classic case there are are no changes to the English, so we can simply write out the original translation, the Japanese version, without looking further down the tree.

Below we list the full rule set description for handling translation synchronization.

Example 5. Combination Grammar for Update of a Translated Document

```
<!-- A= original English
     B= original translation, Japanese
     C= modified English -->
<!-- Deal with paragraphs, which is the unit of translation  -->
<rule-set>
  <element-set match="//para">
    <versionset-a>
      <a-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-a>
    <versionset-b>
      <b-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-b>
    <versionset-c>
      <c-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="new-english"/>
      </c-subtree>
      <include-and-recurse include="no"/>
    </versionset-c>
    <versionset-ab-a-eq-b>
      <ab-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ab-a-eq-b>
    <versionset-ab-a-ne-b>
      <a-subtree include="no"/>
      <b-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ab-a-ne-b>
    <versionset-bc-b-eq-c>
      <bc-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-bc-b-eq-c>
    <versionset-bc-b-ne-c>
      <b-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="old-japanese"/>
      </b-subtree>
      <c-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="new-english"/>
      </c-subtree>
      <include-and-recurse include="no"/>
    </versionset-bc-b-ne-c>
    <versionset-ac-a-eq-c>
      <ac-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-ac-a-eq-c>
    <versionset-ac-a-ne-c>
      <a-subtree include="no"/>
      <c-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-ac-a-ne-c>
    <versionset-abc-a-eq-b-eq-c>
```

```
      <abc-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-b-eq-c>
    <versionset-abc-a-ne-b-ne-c>
      <a-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="old-english"/>
      </a-subtree>
      <b-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="old-japanese"/>
      </b-subtree>
      <c-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="new-english"/>
      </c-subtree>
      <include-and-recurse include="no"/>
    </versionset-abc-a-ne-b-ne-c>
    <versionset-abc-a-eq-b>
      <ab-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="old-english-old-japanese"/>
      </ab-subtree>
      <c-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="new-english"/>
      </c-subtree>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-b>
    <versionset-abc-a-eq-c>
      <ac-subtree include="no"/>
      <b-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-c>
    <versionset-abc-b-eq-c>
      <a-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="old-english"/>
      </a-subtree>
      <bc-subtree include="yes">
        <wrapper-element namespace="translate"
          localname="new-english-old-japanese"/>
      </bc-subtree>
      <include-and-recurse include="no"/>
    </versionset-abc-b-eq-c>
  </element-set>
  <!-- Now deal with all elements above the unit of
         translation elements -->
  <element-set match="*">
    <versionset-a>
      <a-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-a>
    <versionset-b>
      <b-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-b>
    <versionset-c>
```

```
      <c-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-c>
    <versionset-ab-a-eq-b>
      <ab-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ab-a-eq-b>
    <versionset-ab-a-ne-b>
      <a-subtree include="no"/>
      <b-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ab-a-ne-b>
    <versionset-bc-b-eq-c>
      <bc-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-bc-b-eq-c>
    <versionset-bc-b-ne-c>
      <b-subtree include="no"/>
      <c-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-bc-b-ne-c>
    <versionset-ac-a-eq-c>
      <ac-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ac-a-eq-c>
    <versionset-ac-a-ne-c>
      <a-subtree include="no"/>
      <c-subtree include="no"/>
      <include-and-recurse include="no"/>
    </versionset-ac-a-ne-c>
    <versionset-abc-a-eq-b-eq-c>
      <abc-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-abc-a-eq-b-eq-c>
    <versionset-abc-a-ne-b-ne-c>
      <a-subtree include="no"/>
      <b-subtree include="no"/>
      <c-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-abc-a-ne-b-ne-c>
    <versionset-abc-a-eq-b>
      <ab-subtree include="no"/>
      <c-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-abc-a-eq-b>
    <versionset-abc-a-eq-c>
      <ac-subtree include="no"/>
      <b-subtree include="yes"/>
      <include-and-recurse include="no"/>
    </versionset-abc-a-eq-c>
    <versionset-abc-b-eq-c>
      <a-subtree include="no"/>
      <bc-subtree include="no"/>
      <include-and-recurse include="yes"/>
    </versionset-abc-b-eq-c>
  </element-set>
</rule-set>
```

Implementation Details

The primary aim of this paper has been to describe a declarative system for describing Synchronization and merge algorithms on three XML input trees. The need for this system was based on the need to deal with the problems of previous informal approaches and forms a basis for implementation. This section will discuss a number of implementation issues.

Tree Matching

The techniques presented here rely on finding commonality between the three input trees and then assigning version sets to describe the commonality. One metaphor which can be used here is a three dimensional picture of three horizontal trees laid on top of each other and then lines being drawn between the three layers to match up all the common tree nodes.

This matching process is similar to that used in two-way tree comparison, where there are a number of correct solutions. The optimal solution is one with the best matching and is also usually the most compact result [INRIA][DIJK]. Our implementation is based on a derivative of Wu's Longest Common Subsequence (LCS) algorithm applied to trees rather than lists.

We have published details of our n-way representation previously [UNIDELTA] and for this work on synchronization we used this for representing the three data sets merged into one.

Controlling Granularity

In the translation example, we need to implement the concept of a unit of translation, e.g. a paragraph. Our implementation provides this by allowing a different set of rules to be applied to certain elements (those introducing a unit of translation), by specifying those elements in a match XPath, for example for Docbook paras: `<element-rules match="//para">`

In the rules associated with these `para` units it is then possible to handle the subtree as a unit.

Synchronization Annotations

The most basic action is to decide if the node (element, attribute...) is included in the result tree. However, whether or not a node is included in the result tree, the possibilty of inserting other nodes is available. When two subtrees are included, we need a way of distinguishing which is which. Many practical synchronization algorithms do not automatically resolve conflicts and need a way of allowing the user to resolve them. Wrapper elements provide a means for identifying such conflicting subtrees. A number of other options are possible:

- Pre and post comments. Typically used to insert a comment to indicate something has been deleted.
- PCDATA prefix and postfix elements are useful to highlight PCDATA node change and conflicts, for example, an edit1/edit2 conflict could be displayed as: `[e1[original text]][e2[revised text]]` where prefixes were: `[e1[`, `[e2[` and the postfix string was: `]]` in both cases.

To support synchronization annotations we have extended the grammar to include support for them, for example:

```
<versionset-ab-a-ne-b>
  <a-subtree include="no"/>
  <b-subtree include="yes">
    <wrapper-element namespace="dxwarn"
        localname="modifiedEdit1DeletedEdit2"/>
  </b-subtree>
```

```
    <include-and-recurse include="no"/>
  </versionset-ab-a-ne-b>
```

Conclusions

In this paper we have shown that a number of related three-way synchronization problems can be described in a formal grammar and we have presented a sample DTD so that this grammar can be represented in XML. The two examples show how the grammar provides a more formal description of the requirements for particular synchronization scenarios. The grammar also provides the basis for an implementation.

The formal description for how to handle all possible situations in a three-way merge of XML data sets provides a useful descriptive technique for the XML community. The intention is that this will reduce the complexities and cost of automatic merging of XML data sets, and enable many manual, tedious and error-prone processes to be automated.

Bibliography

[BABEL] *Beyond Babel - Simplifying Translation with XML* Robin La Fontaine, Thomas Nichols, Martin Bryan. XML Europe 2004 Conference Paper. April 2004 Available at: http://www.idealliance.org/papers/dx_xmle04/papers/03-03-01/03-03-01.pdf

[UNIDELTA] *Russian Dolls and XML: Handling Multiple Versions of XML in XML* Robin La Fontaine, Thomas Nichols. XML 2003 Conference Paper. December 2003 Available at: http://www.idealliance.org/papers/dx_xml03/papers/05-01-03/05-01-03.pdf

[ORDSYNC] *A 3-way Merging Algorithm for Synchronizing Ordered Trees – the 3DM merging and differencing tool for XML*, Tancred Lindholm, MSc thesis, September 13, 2001. Helsinki University of Technology, Dept. of Computer Science. See: http://www.cs.hut.fi/~ctl/3dm/

[SYNC] *Merging XML files: a new approach providing intelligent merge of XML data sets* Robin La Fontaine. XML Europe 2002 Conference Paper. May 2002 Available at: http://www.idealliance.org/papers/xmle02/dx_xmle02/papers/03-03-04/03-03-04.pdf

[INRIA] *A Comparative Study for XML Change Detection* Verso report number 221, INRIA, 2002. Grégory Cobéna, Talel Abdessalem and Yassine Hinnach. Paper: ftp://ftp.inria.fr/INRIA/Projects/verso/VersoReport-221.pdf

[DIJK] *Structure-Preserving Difference Search for XML Documents* Erich Schubert, Sebastian Schaffert, François Bry. Extreme Markup Languages 2005.August 2005 Available at: http://www.mulberrytech.com/Extreme/Proceedings/html/2005/Schaffert01/EML2005Schaffert01.html