A Delta Format for XML

Identifying changes in XML files and representing the changes in XML

Robin La Fontaine

DELTAXML

Conference Paper: XML Europe

ABSTRACT

This paper describes how changes to XML documents and data files can be represented in XML and proposes a delta format for XML. Although Canonical XML provides a mechanism for verifying that two XML files or documents are equal, it is more often necessary to determine the differences between two XML documents. Such differences should ideally be represented in XML and this paper describes how such changes can be represented in XML with minimal additional attributes and elements.

The paper describes how any changes can be simply represented using this delta format, and how the delta file has the same look and feel as the original files being compared. The paper describes how the delta file can be transformed into HTML for viewing by using a simple XSL style sheet, and by modifying this style sheet changes to particular XML file types can be presented to users in a way that suits their view. In addition, the delta file can be processed by any XML application for other purposes.

The delta format described is applicable to XML files whether or not they have a DTD. If a DTD is available for the input files the paper describes how to determine a DTD for the delta file which enables validation of the changes.

The paper describes an implementation of the delta format and the problem areas encountered when developing the implementation.

CONTENTS

Introduction

Moving data and documents into XML has many advantages, but one disadvantage is that the tools that worked on documents and text to identify and display changes are no longer effective. XML tagged data is neither a simple line-based text file nor a standard document. New tools are needed to identify, intelligently, changes to XML data and documents. New methods are needed to represent these changes. This paper addresses these issues and proposes a method for representing changes to XML using XML itself.

There are many instances where it is necessary to find out what has changed, and these range from basic software testing through to revision control and approval. It is said that death and taxes are the two certainties in life, but change in XML documents is surely a third one. As XML standards emerge, files need to be checked against defined benchmarks. Software that processes XML needs to ensure that the changes made are those that are needed, and that no other changes are made. Updating software to a new version means applying regression tests to check that the results are the same. Revision control means identifying changes and displaying them in a suitable manner for checking and authorisation.

A simple attempt to find out what has been changed between two versions of a document using standard tools quickly demonstrates the basic problems:

Basic problems identifying changes in XML documents

- changes in the order of XML attributes generate spurious changes that should be ignored
- an XML attribute present in one document with a default value but absent and defaulted from the DTD in another document generate spurious changes
- changes in white space within elements generate spurious changes
- it is difficult to know where in the XML tree structure a change has happened when the change is represented by line number
- some elements may appear in any order and so a change to order should not be identified as a change

Ideally, it should be possible to ignore insignificant differences and identify only 'real' changes. These changes should be represented in XML so that all the XML tools can be applied to them, for example to display them or to check them automatically. Canonical XML [CAN-XML] provides a way to ensure two XML files are the same, but provides no help when they are different.

There are other requirements for change representation in XML:

Requirements for Change Representation in XML

1. The change or delta file should be well-formed XML.
2. The delta file should contain all information necessary to identify the changes in both files (and a good test of this is to show that the delta can be combined with either of the files being compared to reproduce the other one).
3. The delta file should not contain information that is unchanged between the two files (a variation of the delta file, discussed below, which does contain all unchanged information is also useful for certain applications).
4. The 'minimum' changes should be represented; this imples that the changes should be identified at the lowest possible level in the XML tree structure.
5. The delta file should look familiar to someone who is familiar with the original data, i.e. the look and feel of the delta file should be the same as the original files. This also ensures that processors designed to process files conforming to the original DTD can easily be modified to work on files conforming to the delta DTD.
6. Knowledge of related XML standards should not be needed to process the change files, i.e. related standards should be used where they are necessary but not otherwise.
7. The delta file should be easy to process using XSL because this is the most suitable tool for transforming it into something else, e.g. into HTML to display.
8. If the files being compared conform to a DTD, then the delta file should also conform to a DTD (it will not be the same DTD). The validation process of the delta file against the delta DTD ensures that the changes

represented in the delta file are, in general, valid changes thus providing a robust change-control representation which is tightly coupled to the original DTD.

9. The delta file should be symmetrical, i.e. it should contain a full record of both the position and content of items added and items deleted. Thus the delta file can be used as a basis for both 'update' and 'undo' operations.

The challenge is to find a way to represent changes to XML data using XML itself in a way that is simple but effective. This paper answers that challenge.

Delta Format Design Approach

Readers of this paper will be aware of the advantage of using XML and therefore familiar with the reasons for choosing to represent the delta file in XML. This approach allows automation, for example to check whether or not the changes are significant, or to sort the changes so that the most important ones are shown first, or simply to summarize or present all the changes in a way that is easy to check.

How, then, can we take two arbitary but similar (they should at least have the same root elements) XML files and represent the differences in a third XML file? The first decision is whether to use new elements to tag the changes or whether to make use of attributes on the existing elements. There is something of a religious war between those who like to use XML attributes to contain their data and those that use XML as a markup language and use tags or elements to delimit data items. Attributes provide order-independence and use less file space, but it is not possible to add attributes to attributes! Attributes are appropriate for meta-data or information that may be applied to many different elements. For this reason, attributes are used as the primary method of identifying why an element is present in the delta file, e.g. because the element has been added or modified. Because it is not possible to add attributes to attributes, a different and weaker mechanism has to be used to represent changes to attributes.

In general it is easier to extend an XML definition that is content-based than one that is attributebased. There are two reasons for this: first, content can be further sub-divided by adding new elements; second, attributes can be added to existing elements to provide further information. Attributes are leaf items that cannot easily be extended in these ways. So it follows that because the delta file is an extension of the basic file it works most naturally for content-based XML files.

To illustrate the approach of using attributes to identify the reason for the inclusion of an element in the delta file, consider the simplest case of an XML file that has no attributes and no DTD. For such a content-based XML file, the basic structure of the delta file can be formed by the addition of a single attribute, the **delta** attribute. This is an optional attribute that takes on one of the values **modify**, **add**, or **delete**. If it is not present, the element is part of the original data in one of the input files, e.g. a sub-element of an element that has been added. This single delta attribute is a simple addition that provides most of the meta-information required in a delta file.

Consider a fragment of XML:

```
<x>
  <y/>
</x>
```

Consider a small change to this:

```
<x>
</x>
```

This can be represented as a delta fragment:

```
<x d:delta="modify">
  <y d:delta="delete"/>
</x>
```

This is a trivial example, but illustrates some of the basic principles. The delta attribute indicates the reason for a particular element being present in the delta file. The basic structure of the delta file is similar to the input files.

To complete the delta representation for content-based XML files, it is necessary to identify changes to PCDATA content and to identify pairs of elements that have been exchanged. Both of these changes are represented in the delta file using additional elements. This in principle is all that is needed, but the precise rules for the representation of delta elements are a little more complex and are described in more detail below.

Changes to attributes that were present in the original files are represented in the delta file in

two new attributes. Each of these lists the names and values of attributes present in one of the original files but changed or absent in the other. Attributes that have not changed do not need to be represented in the delta file.

Delta format for well-formed XML

For well-formed XML, the delta file must be constructed without knowledge of the structure of the file as given in a DTD or XML Schema[1] specification. It is still possible to make a good comparison between similar XML files, and to walk down through the XML tree structure of the two files in a synchronized manner. The elements are assumed to be ordered and the software seeks to find a best match between the sub-elements of any two elements that correspond in the two files. An element in one file is only considered to be a modification of an element in the second file if they are the same type, i.e. elements with the same name space (if any) and local name.

The best match is obtained for elements that are equal, i.e. they have the same name space and local name, their attributes are all equal and their sub-elements are also all equal right down through the tree structure. Such equal elements can provide anchor points in matching the two files and the comparison process can proceed to match elements or text data between these anchor points in the best way to achieve minimum changes in the delta file.

A small number of attributes and elements are used in the delta file to provide all the information needed to represent changes. The attributes are added to existing elements, i.e. elements in the two input files. The additional attributes are:

d:delta[2] to indicate how the containing element has been changed

d:new-attributes and **d:old-attributes** show changes to attributes

The new delta elements are:

d:PCDATAmodify to indicate a change to PCDATA in an element

d:exchange to show one type of element exchanged with another, or an element exchanged with PCDATA

These attributes and elements are described in more detail below.

Special delta attributes

The attribute named **d:delta** can be found on many elements in the delta file. As noted above, it indicates why this element is present in the delta file, e.g. because it has been modified, added or deleted. Starting at the root element of the delta file, this has a **d:delta** attribute which will have a value of "WFmodify" if anything has changed. The 'WF' here means 'Well Formed' to distinguish it from a modification that is based on knowledge of the structure of the DTD. Below this root element, or indeed below any element with a d:delta attribute with value "WFmodify", each element will also have a **d:delta** attribute with one of the following values:

"add" if this element has been added

"delete" if this element has been deleted

"unchanged" if this element is unchanged

"WFmodify" if the attributes and/or content of this element have been modified

There are some constraints on how these are nested as summarised in the table below:

Table 1. Delta attribute values and constraints

d:delta attribute value	Meaning	Constraints
unchanged	Specifies that the element instance is unchanged.	No nested element instances or attributes are needed within the element instance because they have not changed.
add	Specifies that this element instance has been added (present in the 'new' file but not in the 'old')	All the attributes and sub-elements will be included to give a full description of the element that has been added. There will be no d:delta attributes on sub-elements of an

[1]Other XML structure specifications also exist, for example RELAX, TREX or Schematron.

d:delta attribute value	Meaning	Constraints
		element with this d:delta attribute value.
delete	Specifies that this element instance has been deleted (present in the 'old' file but not in the 'new')	All the attributes and sub-elements will be included to give a full description of the element that has been deleted. There will be no d:delta attributes on sub-elements of an element with this d:delta attribute value.
WFmodify	Specifies that this element instance has been modified.	Specifies that this element instance is modified, so all element instances directly within this will have values of "unchanged", "add", "delete" or "WFmodify".

Special delta elements

This section describes the new elements that are introduced into the delta DTD. Where PCDATA items have changed, there will be a **d:PCDATAmodify** element containing the old data within a **d:PCDATAold** element, and the new data within a **d:PCDATAnew** element. When an element in one file is replaced by a different element or PCDATA in the other file, there will be a **d:exchange** element in the delta file. It will contain the data from the first file within a sub-element **d:old** and the data from the second file within a sub-element **d:new**. For a **d:exchange** to occur in the delta file the two items in the input files must be different types, i.e. different element types or one an element and the other a PCDATA item. These special elements are summarised in the table below.

Table 2. Delta element definitions and purpose

Element type name	Content	Purpose
PCDATAmodify	(PCDATAold, PCDATAnew)	Records a change to PCDATA content
PCDATAold	#PCDATA	Records the original PCDATA content
PCDATAnew	#PCDATA	Records the new PCDATA content
exchange	(old, new)	Records a change where one element or PCDATA has been exchanged for another element or PCDATA
old	ANY	Records the original element or PCDATA content
new	ANY	Records the new element or PCDATA content

Delta format for changes in attributes

Changes to XML attributes need to be detected and represented in the delta file. Any changes to attribute values are represented using two additional attributes in the delta file: d:old-attributes and d:new-attributes.

The first of these, d:old-attributes, contains all values of attributes that appeared in the old file and which have either been changed or deleted in the new file. The second, d:new-attributes, contains all values of attributes that appear in the new file and which have either been changed or added in the new file. Attributes that have not changed are not included in the delta file, except in the special case, discussed later, where they form a part of the key to an element.

The value of these delta attributes is a simple encoding of all the attribute values changed,

added or deleted. For example, an attribute named foo that has been changed will be included as “foo=’oldvalue’ “ within the delta attribute d:old-attributes, and as “foo=’new-value’ “ within the delta attribute d:new-attributes. A deleted attribute will appear only in d:old-attributes. An added attribute will appear only in d:new-attributes. The character used to delimit attribute values will generally be a double quote, represented as the entity " , a single quote or a verticle bar. The character is picked according to the content of the attribute value, e.g. if it contains a double quote then this cannot be used as a delimiter. The example below shows that on an element a, the value of the attribute href has been changed from href=’www.deltaxml.com’ to href=’http://www.deltaxml.com’. In addition, the attribute xx=’value of xx’ has been deleted and the attribute yy=’value of yy’ has been added.

```
<a d:old-attributes= “href=’www.deltaxml.com’
xx=’value of xx’”
  d:new-attributes= “href=’http://www.deltaxml.
  com’ yy=’value of yy’”/>
```

A number of other ways of handling changed attribute values was considered, but it seemed most appropriate to use attributes to handle these changes. The string handling capabilities of XSL make it reasonably easy to manipulate these composite attributes encoded within another attribute value.

Delta Format for XML with DTD

Although XML documents and files can be compared as well-formed XML, more can be achieved if the comparator has some knowledge of the structure of the data.

The Influence of DTDs

DTDs provide XML with entity expansion details, default attribute values and some structural validation. For XML applications, the entity expansion and default attribute values are necessary to achieve the basic data on which an application will run, and the Infoset specification [INFOSET] details what this is. However, it is possible to use knowledge of the allowed structure of an XML file, as specified in the DTD, to provide a more intelligent comparison which will typically result in smaller and more accurate delta files.

The DTD will specify which elements are required, optional and/or repeated. With knowledge of this, a comparator can work out more accurately how the two files being compared correspond with one another. For example, consider the two fragments of XML:

```
<fragment>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <address>21 High Street</address>
  <address>Malvern</address>
  <address>Worcester</address>
</fragment>
```

and

```
<fragment>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <firstName>Mike</firstName>
  <lastName>Jones</lastName>
  <address>Worcester</address>
</fragment>
```

A direct comparison of these two, without knowledge of the structure, might yield the following delta file[2]:

```
<fragment d:delta=”WFmodify”>
  <firstName d:delta=”unchanged”/>
  <lastName> d:delta=”unchanged”/>
  <d:exchange>
    <d:old>
      <address>21 High Street</address>
    </d:old>
    <d:new>
      <firstName>Mike</firstName>
    </d:new>
  </d:exchange>
  <d:exchange>
    <d:old>
      <address>Malvern</address>
    </d:old>
    <d:new>
      <lastName>Jones</lastName>
    </d:new>
  </d:exchange>
  <address delta=”unchanged”/>
</fragment>
```

However, the above is obviously not correct. It is ‘obvious’ to us only because of the element names and our interpretation of these: if the element names or values had been different it would not have been ‘obvious’ that it was incorrect. It is only possible to determine what is correct with some knowledge of the DTD. Without a DTD, a

[2]In this and other examples, a d: prefix is used to show elements and attributes that belong to the deltaxml namespace. The software uses a deltaxml: prefix for this.

comparator can only assume that the structure is:

```
<!ELEMENT fragment (firstName | lastName |
address)*>
```

and in that case the above delta is reasonable. But if the definition of the element fragment was, as is more likely:

```
<!ELEMENT fragment (firstName, lastName,
address*)*>
```

then the comparator can make a better job of working out the changes. With knowledge of this DTD, it is possible to work out that the address elements have been deleted rather than changed to a different type of element. So a more accurate delta file can be created:

```
<fragment d:delta="modify">
  <firstName d:delta="unchanged"/>
  <lastName d:delta="unchanged"/>
  <address d:delta="delete">21 High
  Street</address>
  <address d:delta="delete">Malvern</
  address>
  <address d:delta="delete">Worcester</
  address>
  <firstName d:delta="add">Mike</firstName>
  <lastName d:delta="add">Jones</lastName>
  <address d:delta="add">Worcester</
  address>
</fragment>
```

It is evident that this delta file is more meaningful and correct. It could, for example, be displayed to show the changes much more easily than the original delta. The non-DTD-aware delta file would be useful to check whether two files were the same or to provide an update file for changes. Although being DTD-aware provides a better result, for some DTDs the basic comparison without knowledge of the DTD is adequate. This is the case when the DTD is relatively unstructured and the data is all order-significant.

Requirements for a delta DTD

The delta file format for a delta file that is created with knowledge of the structure of the DTD is similar to the delta file for well-formed XML, with some additional elements and attribute values. In order to understand these differences, it is worth considering first how a normal DTD is modified to form a delta DTD, bearing in mind the requirements outlined earlier in this paper, and the specific requirements below that the delta DTD should meet.

Delta DTD requirements

1. The delta DTD should be a superset of the original DTD, i.e. all XML files that conform to the original DTD shall also conform to the delta DTD.
2. The delta DTD should provide as much validation as possible for the delta file, e.g. changes to a choice should limit the change to be elements that are valid within that choice.
3. The delta DTD should introduce as few additional elements as possible. This is to keep the delta DTD as simple and small as possible.
4. The delta DTD should be able to represent all possible changes between XML files that conform with the original DTD.
5. The delta DTD should take account of the fact that specified repeating elements or groups may occur in the XML files in any order, i.e. they are order-independent.
6. The delta DTD should allow order-independent elements to have keys which uniquely identify the elements within the context of their parent element, i.e. the value of the key is unique for all subelements of a given type within a repeating group.
7. For any given element structure, there should be only one delta structure for that element and there should be a deterministic way of transforming from the original content particle structure to the delta content particle structure.

These requirements present something of a challenge. To take an arbitarily-complex element definition and create a new element definition that permits all possible changes is not simple. In addition, to be useful for XML data files, as opposed to document files, this transformation should take account of the order-independence of some sub-elements and keys to these sub-elements. It is not possible within the constraints of this paper to cover the full details of how this is achieved, but a number of examples will be presented showing how a DTD is transformed into a delta DTD.

Transforming a DTD to a delta DTD

This section describes how a given DTD can be transformed into a delta DTD. For example, a DTD

that represents a parts list can be transformed into a DTD that represents changes to a parts list. Or a DTD for a delivery schedule can be tranformed into a DTD for changes to a delivery schedule; a DTD for XML Schema can be transformed into a DTD representing changes or updates to a schema. These delta DTDs provide structure and validation for the delta file, but they are often also useful in their own right if one of the requirements for an XML format is to perform or represent updates. The following are examples of changes that may be made to the element type declarations of a DTD with reference to their content.

Delta DTD for PCDATA elements

Elements whose content is #PCDATA will have their content changed from:

(#PCDATA) or (#PCDATA)*

to (#PCDATA | d:PCDATAmodify)*

For example, the element type declaration:

```
<!ELEMENT comment (#PCDATA)>
```

would become:

```
<!ELEMENT comment (#PCDATA |
d:PCDATAmodify)* >
```

The representation of the differences between:

```
<comment>This is a comment</comment>
```

and:

```
<comment>This is another comment</comment>
```

would be represented in the delta file as:

```
<comment d:delta=”modify”>
  <d:PCDATAmodify>
    <d:PCDATAold>This is a comment</
    d:PCDATAold>
    <d:PCDATAnew>This is another comment</
    d:PCDATAnew>
  </d:PCDATAmodify>
</comment>
```

It is also possible to use the structure for a finer-grained change, for example:

```
<comment d:delta=”modify”>This is
  <d:PCDATAmodify>
    <d:PCDATAold>a</d:PCDATAold>
    <d:PCDATAnew>another</d:PCDATAnew>
  </d:PCDATAmodify> comment
</comment>
```

Delta DTD for MIXED elements

MIXED content elements can be treated in a similar manner to #PCDATA. Note that the items within the exchange element may have elements as well as PCDATA in them. For example:

```
<!ELEMENT mixedContentExample (#PCDATA | a
| b)*>
```

would become:

```
<!ELEMENT mixedContentExample (#PCDATA | a
| b | d:exchange | d:PCDATAmodify)*>
```

Delta DTD for required element

In the case of a single required element within another element an example of an element declaration is:

```
<!ELEMENT ex1 (x)> <!ELEMENT x (#PCDATA)>
```

The resultant delta element declaration is:

```
<!ELEMENT ex1(x)?>
<!ATTLIST ex1 d:delta (modify | unchanged
| originalData) #IMPLIED>
<!ELEMENT x (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST x d:delta (modify | unchanged |
originalData) #IMPLIED>
```

In the above, as x is required it cannot be added or deleted and so the delta attribute does not need to allow these values, although in general all values are allowed because the element **x** may be used in other elements. The delta element declaration makes **(x)** optional so that in the case where **ex1** is unchanged, no data need be included within it.

For example:

```
Input file 1:
<ex1><x>a b c</x></ex1>

Input file 2:
<ex1><x>x y z</X></ex1>
```

Delta file element:

```
<ex1 d:delta=”modify”>
  <x d:delta=”modify”>
    <d:PCDATAmodify>
```

```
    <d:PCDATAold>a b c</d:PCDATAold>
    <d:PCDATAnew>x y z</d:PCDATAnew>
   </d:PCDATAmodify>
  </x>
</ex1>
```

Delta DTD for repeated elements

Consider next the simple case of a repeated optional element within another element. The repeated element is ordered, so successive pairs are compared. Example element declaration

```
<!ELEMENT ex3 (x*)> <!ELEMENT x (#PCDATA)>
```

The corresponding delta DTD fragment would be:

```
<!ELEMENT ex3(x*)>
<!ATTLIST ex3 d:delta (modify | unchanged
| originalData) #IMPLIED>
<!ELEMENT x (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST x d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
```

An example of this operating on two files is shown below.

```
Input file 1:
<ex3><x>a b c</x><x>x y z</x></ex3>

Input file 2:
<ex3><x>a b c</x><x>!x y z!</x><x>c d e</
x></ex3>
```

And the result is:

```
<ex3 d:delta="modify">
  <x d:delta="unchanged"></x>
  <x d:delta="modify">
    <d:PCDATAmodify>
      <d:PCDATAold>x y z</d:PCDATAold>
      <d:PCDATAnew>!x y z!</d:PCDATAnew>
    </d:PCDATAmodify>
  </x>
  <x d:delta="add">c d e</x>
</ex3>
```

Delta DTD for repeated, unordered elements

It is interesting to consider the same case of a repeated optional element within another element, but where the order of x is not significant.

```
<!ELEMENT ex4 (x*)>
<!ATTLIST ex4 delta (modify | unchanged |
originalData) #IMPLIED>
<!ELEMENT x (#PCDATA)>
<!ATTLIST x delta (add | delete |
originalData) #IMPLIED>
```

There are some differences here compared to the ordered case. Because the instances of x are not in order, there is no way to work out which ones correspond, it is only possible to say which ones are present in both files and which are not. Therefore it is not possible to "modify" any particular instance, it is only possible to identify instances that have been added and ones that have been deleted. Occurrences that have not changed will not appear in the delta file, so there is no possibility of having an instance of x with a delta attribute of "unchanged".

```
Input file 1:
<ex4><x>a b c</x><x>x y z</x></ex3>

Input file 2:
<ex4><x>!x y z!</x><x>a b c</x><x>c d e</
x></ex3>
```

The result is shown below. Note that <x>a b c</x> is in both input files but does not appear at all in the delta file. There are no changes, only additions and deletions.

```
<ex4 d:delta="modify">
  <x d:delta="delete">x y z</x>
  <x d:delta="add">!x y z!</x>
  <x d:delta="add">c d e</x>
</ex4>
```

Delta DTD for keyed, repeated, unordered elements

The next step is to consider a similar example, but where the repeated unordered sub-element has a key which identifies it within the context of its parent element. This example presents the case of a keyed or named element which is repeated within another element. The keyed element is the element **namedElement** and its key is the attribute **attr1**. Therefore any two **namedElement** instances which have the same value for this attribute will be taken to be corresponding elements in the two files, and so they can be compared.

```
<!ELEMENT ex16 (namedElement*)>
<!ELEMENT namedElement (x)>
<!ATTLIST namedElement
  attr1 CDATA #REQUIRED
  attr2 CDATA #REQUIRED>
<!ELEMENT x (#PCDATA)>
```

The resultant delta DTD fragment would be:

```
<!ELEMENT ex16 (namedElement*)>
<!ATTLIST ex16 d:delta (modify | unchanged
| originalData) #IMPLIED>
<!ELEMENT namedElement(x)?>
<!ATTLIST namedElement
  attr1 CDATA #IMPLIED
  attr2 CDATA #IMPLIED
  d:delta (add | delete | modify |
  unchanged | originalData) #IMPLIED
  d:new-attributes CDATA #IMPLIED
  d:old-attributes CDATA #IMPLIED>
<!ELEMENT x (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST x d:delta (modify | unchanged |
originalData) #IMPLIED>
```

In this delta DTD, the required attributes **attr1** and **attr2** on **namedElement** have been changed to be implied attributes, because any attribute that does not change is not included in the delta file. This would cause a validation error if the attributes remained required. In fact, because attr1 is part of the key, it will always be present in the delta file in this case. However, **namedElement** could be used elsewhere in the DTD as an unnamed element or even with **attr2** as the key, and so **attr1** should be implied and not required. The same argument, possible use elsewhere in the DTD, also applies to making the content of **namedElement** optional, although there is another reason for this here: if only an attribute in **namedElement** changes, then its content will be empty because it is unchanged. There are many different factors to be taken into consideration in generating a delta DTD. Consider an example based on this DTD fragment.

```
Input file 1:
<ex16>
  <namedElement attr1="a" attr2="z">
    <x>a b c</x>
  </namedElement>
  <namedElement attr1="b" attr2="z">
    <x>d</x>
  </namedElement>
  <namedElement attr1="c" attr2="z"/>
    <x>e</x>
  </namedElement>
</ex16>

Input file 2:
<ex16>
  <namedElement attr1="b" attr2="z">
    <x>d</x>
  </namedElement>
  <namedElement attr1="c" attr2="z">
    <x>e f</x>
  </namedElement>
  <namedElement attr1="d" attr2="z">
    <x>a b c</x>
  </namedElement>
</ex16>
```

Matching up corresponding **namedElements** in the two files gives the following delta file:

```
<ex16 delta="modify">
  <namedElement attr1="c" d:delta="modify">
    <x d:delta="modify">
      <d:PCDATAmodify>
        <d:PCDATAold>e</d:PCDATAold>
        <d:PCDATAnew>e f</d:PCDATAnew>
      </d:PCDATAmodify>
    </x>
  </namedElement>
  <namedElement attr1="a" attr2="z"
  d:delta="delete">
    <x>a b c</x>
  </namedElement>
  <namedElement attr1="d" attr2="z"
  d:delta="add">
    <x>a b c</x>
  </namedElement>
</ex16>
```

In the above example, there are three **namedElement** instances in both files, but looking at the keys it is clear that "a" has been deleted, "d" added and "c" has been modified as shown in the delta result. For the **namedElement** with **attr1="c"**, the attribute **attr2** does not appear because it is unchanged, but **attr1** must appear because it is part of the name and so needed for correct navigation through the delta file.

Delta DTD for sequence content particles

The next example considers the changes required to convert a sequence into the delta DTD. A sequence may contain required or optional elements or complex content particles, which may be repeated. Consider the following example:

```
<!ELEMENT ex6 (x, (y, z)?, x*)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT z (#PCDATA)>
```

This would be converted to a delta DTD as follows:

```
<!ELEMENT ex6 (x,(y, z)?, x*)?>
<!ATTLIST ex6 d:delta (modify | unchanged
| originalData) #IMPLIED>
<!ELEMENT x (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST x d:delta (add | delete | modify
```

```
| unchanged | originalData) #IMPLIED>
<!ELEMENT y (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST y d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
<!ELEMENT z (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST z d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
```

In this case the only change to the structure of the element declaration is that it is made optional to cater for the unchanged case. An example of some input data and a resulting delta file is shown below.

```
Input file 1:
<ex6><x>a b c</x></ex6>

Input file 2:
<ex6><x>a b c</x><y>a</y><z>b</z></ex6>
```

The resulting delta file is as follows:

```
<ex6 d:delta="modify">
  <x d:delta="unchanged></x>
  <y d:delta="add">a</y>
  <z d:delta="add">b</z>
</ex6>
```

Delta DTD validation for the exchange of elements

A set of three element types denotes exchanges of element instances within a document. An exchange is the replacement of zero or more element instances, which conform to a content particle in the element declaration, with another list of zero or more element instances conforming to the same content particle definition. An exchange sequence is represented as:

```
<d:exchangeStart/>
xx
<d:exchangeMiddle/>
yy
<d:exchangeEnd/>
```

where **xx** is the data for zero or more element instances from the first file, and **yy** the data for the same content particle in the second file. Each element within **xx** will have a delta attribute set to “delete” and each element within yy will have a delta attribute set to “add”.

This structure allows a DTD to be transformed into a delta DTD in a way that preserves much of the validation capabilities of the original DTD. Some examples are shown below to illustrate this.

Delta DTD for choice content particle

The next example considers the changes required to convert a required choice content particle into the delta DTD. A required choice is a choice between elements or content particles where one of the items must be present. Consider the following example:

```
<!ELEMENT ex10 (x | y | z)>
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
<!ELEMENT z (#PCDATA)>
```

This would be converted to a delta DTD as follows:

```
<!ELEMENT ex10 (x | y | z |
(d:exchangeStart, (x | y | z),
  d:exchangeMiddle, (x | y | z),
  d:exchangeEnd))?>
<!ATTLIST ex10 d:delta (modify | unchanged
| originalData) #IMPLIED>
<!ELEMENT d:exchangeEnd EMPTY>
<!ELEMENT d:exchangeMiddle EMPTY>
<!ELEMENT d:exchangeStart EMPTY>
<!ELEMENT x (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST x d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
<!ELEMENT y (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST y d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
<!ELEMENT z (#PCDATA | d:PCDATAmodify)*>
<!ATTLIST z d:delta (add | delete | modify
| unchanged | originalData) #IMPLIED>
```

This is a more complex structure in the delta DTD, but it preserves much of the validation of the original DTD. An exchange of choice items must be between valid elements. An example of some input data and a resulting delta file is shown below.

```
Input file 1:
<ex10><x>a b c</x></ex10>

Input file 2:
<ex10><y>a b c</y></ex10>
```

The resulting delta file is as follows:

```
<ex10 d:delta="modify">
  <d:exchangeStart/>
  <x d:delta="delete">a b c</x>
  <d:exchangeMiddle/>
  <y d:delta="add">a b c</y>
  <d:exchangeEnd/>
</ex10>
```

Delta DTD for other combinations of content particle

Many more examples could be given for different possible combinations. The methodology does cater for arbitarily-complex content particles. Typically, the more complex the original content particle the more complex the delta will be.

Industrial applications

There are a number of obvious applications of DeltaXML, for example in software regression testing and revision control. However, the availability of changes between XML documents and data in XML enables new applications and automation to be developed. These are probably only limited by imagination. If applications that read or write large XML files are adapted to read or write the delta format, then revisions can be processed more quickly and communication between applications can be made faster.

A variation on the delta file is to include not only the changes but also the original data. This makes it very easy to generate electronic documents that have changes represented in them, something that is not otherwise easy with XML.

A Transformation API for XML (TrAX) interface is also available so that the delta engine can be used in a pipeline of XML applications, processing fragments or complete files as required and connected by SAX to input and output processes.

There are always comparison requirements that cannot be met by a standard comparator. Many of these can be satisfied by adding an XSL filter to the input files and/or to the output file. This would, for example, allow XML comments to be changed into markup and thus compared, and then turned back into comments on output. Processing instructions could be handled in a similar manner.

Implementation Issues

This section outlines some of the issues that were encountered during implementation of the software.

The development of the method for representing changes, the delta DTD, and the way in which any DTD can be transformed into a delta DTD, has taken some years and been through many stages of design[3]. For example, an initial design generated separate elements for add, delete and modify, i.e. up to three elements for each element in the original DTD. This gave good control but resulted in an excessive number of elements, especially for engineering data DTDs with several thousand elements in the original DTD.

It was one of the inital goals that complex DTDs and large files could be handled. Delta DTDs have been generated for DTDs with up to 2000 elements and files up to 35Mb have been compared. Because of the need to read large files, and because the work was started in the early days of XML, the DOM was not used as an internal data structure. Instead a special array representation was developed, in Java, to provide an economical in-memory image of the XML information set. This array structure also has the additional advantage that files can be read in more quickly. It has not been difficult to add SAX and DOM interfaces to this to provide an embedded Java process that can be pipelined with other XML processes, e.g. using TrAX. Integrated XSL filters for input and output can also be provided.

Two of the most problematic areas have been the handling of white space and the DOCTYPE. These simple issues have proved to be difficult to handle in a manner that is consistent and adaptable. For example, the DOCTYPE statement has a file reference but it is difficult to generate this without knowledge of the file system on which the application is running, and the local conventions in use. Configuration files can solve these problems but they make the system more complicated. For these reasons, and because all entities are expanded and attribute default values are incorporated at the input stage, a DOCTYPE does not need to be provided in the output files.

Conclusions

This paper demonstrates that it is possible to represent arbitary changes to an XML document in XML with the addition of a few simple attributes and elements. This has the considerable advantage that the changes can be processed using any available XML technology, for example XSL can be used to provide displays of the changes.

It has also be shown that knowledge of the structure of an XML document, in the form of the DTD, can be used to provide a more intelligent representation of the changes. Any arbitarily complex DTD can be transformed into a delta DTD by applying transformations to the content particle structure of each element in the DTD and adding some additional attributes and elements to the DTD.

[3]Patent applied for.

Further details and implementations of this technique are available, http://www.deltaxml.com. Implementations include software for comparing well-formed XML documents and DTD-aware comparators for some DTDs. DTD-aware comparators for other DTDs can be developed but this is not yet a fully automated process.

Bibliography

[CAN-XML] CAN-XML. Canonical XML Version 1.0, Latest version: http://www.w3.org/TR/xml-c14n.

[INFOSET] INFOSET. XML Information Set, Latest version: http://www.w3.org/TR/xml-infoset.

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