*This is a draft for a chapter in the 5th edition of **The XML Handbook**, due for publication in late 2003.*

Authors: Martin Bryan, Robin La Fontaine

Change Management for XML, in XML

- The benefits of change management
- Identifying changes
- The effects of structure on changes
- Using XML to record changes
- The benefits of exchanging delta files
- DeltaXML – Understanding XML changes

As production deadlines become tighter and tighter, the time available for checking changes to published data is being dramatically reduced. Whether the data to be published has been created or edited by humans, computer programs or database reporting systems, the changes that have taken place during a specific part of the production cycle need to be identified and validated. As more and more data is generated, exchanged and processed using XML the need to identify changes within XML files is becoming widely recognized. But the structured nature of XML markup means that the rules needed to identify changes within XML files are radically different from those used to identify changes in unstructured text files.

In this chapter we will explain the benefits of using XML to identify differences between XML files, and look at how the combination of an XML-based differencing engine and XSLT-based data transformation tools can simplify and speed up production cycles.

1.1 The benefits of change management

Before taking a look at how you can identify differences in XML files you need to understand the benefits that the introduction of a change management systems can bring. An XML-based change management system can be used for a wide variety of purposes, including:

- *Auditing editorial processes* by allowing editors to check changes made against the original data.
- *Managing data translation* by identifying sections of text that need to be translated within updated versions of a master document.
- *Synchronizing changes* made on disparate systems by automatically merging changes directly into the original XML file.
- *Reducing processing* by exchanging only changed data. By pre-processing data feeds, database repopulation costs can be reduced.
- *Reducing bandwidth* by transmitting only updates. Clients can merge these updates with their existing data records to keep local copies of the master database up-to-date.
- *Increasing security* by splitting sensitive data into separate streams which are reassembled on delivery.

By providing a way to intercept a stream of XML data, compare it with the original data, and remove any data that has not changed you can reduce both processing time and storage space. A *delta file* recording the differences between two files can easily be converted to HTML for viewing in a browser as part of the revision management process. Colour and text characteristics can be used to highlight changes so that reviewers can see exactly has been added, modified or deleted.

The granularity of changes is often important. Sometimes it is not advisable to modify part of an object, such as a single field in a record or table: instead the system must ensure that the whole set of related fields are updated at the same time. Differencing systems that understand the structure of the underlying data are able to manage the granularity of data updates efficiently.

Data in multilingual sites is often represented in XML. The initial translation of the source material is relatively straightforward, but major problems are introduced when a multilingual site is updated without efficient change management systems being in place. In order to update the translations, it is necessary to know exactly which changes have been made to the original text. As you only need to update those parts of the files that have been changed in the source data, creating a delta file of the changes, at a suitable level of granularity, translating these and then merging the changes with the original translation will reduce significantly the time and cost of translation.

Exchanging delta files rather than retransmitting large amounts of data can significantly reduce bandwidth requirements of web services based on XML. A small change at one end of the pipeline can be sent to the other end as a delta file. If it has been recorded in XML, the delta file can be encapsulated directly within a SOAP envelope.

File differencing techniques can also be used to allow remote users to update a database which is represented as an XML file. The ability to merge changes from multiple sources into a single file is, however, key to many applications, not just those related to database publishing. Typically, however, differencing engines cannot identify when records have simply been reordered, rather than changed. By providing individual sets of fields with unique keys it is possible to restrict the reporting of changes to just the additions and deletions to the file, without noting changes caused by the reordering of entries.

1.2 Identifying changes

What constitutes a change to an XML file, and what doesn't?

When comparing XML files it is important not to look simply at the sequence of characters that make up the recorded file. For example, the following files are, in fact, identical as far as XML is concerned, though a string-based differencing engine would falsely identify a large number of differences:

***Example 0-1* No spaces or namespaces**

```
<record xmlns="http://www.myco.com/records" id="b123">
<name>Michael Brown</name><born>1984-03-08</born><sex>M</sex>
</record>
```

***Example 0-2* With spaces and namespaces**

```
<staff:record id="b123"
 xmlns:staff="http://www.myco.com/records">
  <staff:name>Michael Brown</staff:name>
  <staff:born>1984-03-08</staff:born>
  <staff:sex>M</staff:sex>
</staff:record>
```

The second file differs from the first in a number of respects, including:

- Whitespace has been inserted within and between markup tags to indent embedded start tags and to place elements on separate lines.
- The order of the attributes has been changed.
- In the second example, a specific namespace has been assigned to all elements, whereas the first example used the default namespace.

None of these changes has any significance as far as comparing the information set created by the two XML files is concerned.

The *Canonical XML* specification published by the World Wide Web Consortium (`http://www.w3.ort/TR/xml-c14n`) is designed to let files to be converted to a standard format prior to being compared. This ensures that documents have a standard encoding (UTF-8), normalized line breaks (using `#xA` only), alphabetically ordered normalized attributes that are always enclosed in double quotes, including all default attributes assigned by the DTD, the replacement of CDATA, special character references and parsed entities by their character content, empty elements converted to start-tag/end-tag pairs, superfluous namespace declarations removed with all namespace definitions alphabetically ordered, whitespace outside the document element removed but any inside nested elements retained (other than superfluous carriage returns), and details of the XML declaration and DTD removed. By adopting this canonicalized format for comparing files it becomes easier to identify the first point at which the character string changes, but this does not help programs to identify all changes made to the structure or contents of the two representations.

When comparing XML files it is important that you are able to identify each of the following types of changes:

- Changes in the contents of parsed data.
- Changes in the values of attributes.
- The deletion of markup components (elements or attributes).
- The addition of new markup components.

1.3 The effects of structure on changes

Some differences are significant, but not as significant as a simple string differentiation technique would suggest. Consider the following pair of examples:

Example 0-3 Original records

```
<record id="b123">
  <name>Michael Brown</name>
  <born>1984-03-08</born>
  <sex>M</sex>
</record>
<record id="b124">
  <name>Gillian Bryan</name>
  <born>1951-03-06</born>
  <sex>F</sex>
</record>
```

Example 0-4 Updated records

```
<record id="b124">
  <employee-no>BR12</employee-no>
  <name>Gillian Bryan</name>
  <born>1951-03-06</born>
  <sex>F</sex>
</record>
<record id="b123">
  <employee-no>BR24</employee-no>
  <name>Michael Brown</name>
  <born>1984-03-08</born>
  <sex>M</sex>
</record>
```

An XML-aware differencing engine should be able to identify when a new element, such as the `employee-no` element introduced into the updated examples, has been added to a nested element set, but that the remainder of the nested elements are identical to the source file.

Ideally, XML-aware differencing engines should also allow users to indicate whether the order of the repeatable elements is or is not significant. For example, if the order is significant then there is a significant mismatch between the two files shown above, but if order is not significant the only change is the addition of the two new elements.

Identifying the minimal set of actual changes that have occurred in an XML file is a non-trivial task because it involves look-ahead. When you find a change in the sequence of elements you need to search the rest of the data stream to identify a possible re-synchronization point. This may not occur at an element boundary, as the following examples show:

***Example 0-5* No embedded elements**

```
<html>
<body>
<p>This paragraph has special text</p>
</body>
</html>
```

***Example 0-6* With embedded elements**

```
<html>
<body>
<p>This paragraph has <em>special</em> text</p>
</body>
</html>
```

The parsed contents of the two paragraph elements in these XML files is identical, but the structure of the files is different. The differencing engine has to determine whether or not the change of emphasis for one word in the text in any way changes the contents of the containing element. Example 1.7 shows how the DeltaXML suite from Monsell EDM Ltd differentiates words that remain part of the unchanged paragraph text from those that have been changed, irrespective of whether the changes are structural or just textual.

***Example 0-7* DeltaXML representation of textual differences**

```
<html
 xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
 deltaxml:delta="WFmodify">
 <body deltaxml:delta="WFmodify">
  <p deltaxml:delta="WFmodify">
   This paragraph has
   <deltaxml:exchange>
    <deltaxml:old>special</deltaxml:old>
    <deltaxml:new>
     <em deltaxml:delta="add">special</em>
    </deltaxml:new>
   </deltaxml:exchange>
   text
  </p>
 </body>
</html>
```

1.4 Using XML to record changes

There are many ways in which changes to files can be recorded. Simple differencing tools typically record the number of the character or line at which the change has been identified, with a second number identifying the length of the changed data, or the point at which re-synchronization occurs. These two counts can then be associated with the data that is to replace the existing data between the two points, if any. While such information is adequate for recording changes internally so that they can be displayed on screen as long as the pointers are retained in memory, they are not suitable for long-term storage of information because they make interpretation of the data dependent on access to the original software.

An alternative approach is to use a standardized object referencing system, such as those provided by the XML Pointer (XPointer) and XML Path (XPath) specifications, to identify the points at which changes have been made to the source file. Such systems, however, run into difficulty if they are required to apply multiple changes to a file if the first of a set of changes results in a change to the element structure. For example, a change to the word `special` in Example 1.5 requires a reference to a substring in an element using a path statement of the form `xhtml/body/p[(substring(text(),19,7)]/` whereas for the updated file shown in Example 1.6 a reference to the same word will be take the form `xhtml/body/p[1]/em`. If the changes have been made by multiple editors, based on the contents of the source file, the sequence in which the changes are made will affect the resultant file because the numeric positions in the substring have changed.

Differencing engines typically only record the place at which the change is to occur, and the text that is to be used in that position for future versions of the text. This makes it very difficult to both verify that the change is correct and to audit the change at a later date. For example, the proposed XUpdate XML Update Language specification (`http://www.xmldb/org/xupdate/xupdate-wd.html`) defines a set of XML elements that are designed to allow users to record insertions, additions and removals of elements in an XML tree. It uses XML Paths to identify the points in the source document at which changes are to be made, using elements named `insert-before`, `insert-after`, `append`, `update`, `remove`, `rename`, `variable`, `value-of` and `if`. The following examples show how a file can be updated using the XUpdate language.

Example 0-8 Input data

```
<?xml version="1.0"?>
 <addresses version="1.0">
  <address id="1">
   <fullname>Andreas Laux</fullname>
   <born day='1' month='12' year='1978'/>
   <town>Leipzig</town>
   <country>Germany</country>
  </address>
 </addresses>
```

Example 0-9 XUpdate modifications

```
<?xml version="1.0"?>
 <xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/addresses/address[1]" >
   <xupdate:element name="address">
    <xupdate:attribute name="id">2</xupdate:attribute>
     <fullname>Lars Martin</fullname>
     <born day='2' month='12' year='1974'/>
     <town>Leizig</town>
     <country>Germany</country>
    </xupdate:element>
   </xupdate:insert-after>
 </xupdate:modifications>
```

***Example 0-10* Modified file**

```
<?xml version="1.0"?>
 <addresses version="1.0">
  <address id="1">
   <fullname>Andreas Laux</fullname>
   <born day='1' month='12' year='1978'/>
   <town>Leipzig</town>
   <country>Germany</country>
  </address>
  <address id="2">
   <fullname>Lars Martin</fullname>
   <born day='2' month='12' year='1974'/>
   <town>Leizig</town>
   <country>Germany</country>
  </address>
 </addresses>
```

DeltaXML takes an alternative approach, recording both the text that has been removed and the text that is to replace it at the relevant point in the tree of a delta file. For example, given the following source file:

***Example 0-11* Original file**

```
<doc>
 <title>Identifying changes in XML files</title>
 <author>Robin La Fontaine</author>
 <date>2003</date>
</doc>
```

and a modification to it so that it reads:

***Example 0-12* Modified file**

```
<doc>
 <title>Identifying changes in XML files</title>
 <editor xmlns="http://www.phptr.com">Charles F. Goldfarb</editor>
 <date>2003</date>
</doc>
```

the output produced by the default setting of DeltaXML will only show the pair of exchanged elements:

***Example 0-13* DeltaXML standard change report**

```
<doc
 xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
 xmlns:p0="http://www.phptr.com" deltaxml:delta="WFmodify">
 <title deltaxml:delta="unchanged" />
 <deltaxml:exchange>
  <deltaxml:old>
   <author deltaxml:delta="delete">Robin La Fontaine</author>
  </deltaxml:old>
  <deltaxml:new>
   <p0:editor deltaxml:delta="add">Charles F. Goldfarb</p0:editor>
  </deltaxml:new>
 </deltaxml:exchange>
 <date deltaxml:delta="unchanged" />
</doc>
```

DeltaXML can also be configured to provide a change record that contains all of the unchanged elements in a well-formed XML file, as well as details of the changes. In the case of the above examples this form of output takes the form:

***Example 0-14* DeltaXML all data report**

```
<doc
 xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
 xmlns:p0="http://www.phptr.com" deltaxml:delta="WFmodify">
 <title deltaxml:delta="unchanged">Identifying changes in
  XML files</title>
 <deltaxml:exchange>
  <deltaxml:old>
    <author deltaxml:delta="delete">Robin La Fontaine</author>
  </deltaxml:old>
  <deltaxml:new>
   <p0:editor deltaxml:delta="add">Charles F. Goldfarb</p0:editor>
  </deltaxml:new>
 </deltaxml:exchange>
 <date deltaxml:delta="unchanged">2003</date>
</doc>
```

The advantage of this approach is that users can, using a simple XSLT transformation, convert this XML file into a displayable HTML file, or a printable file, in which both the original text and the new text can be displayed side-by-side, e.g.

Title: *Identifying changes in XML files*
~~**Author:** Robin La Fontaine~~ **Editor:** Charles F. Goldfarb
Date: 2003

Changes to attributes can be recorded in a similar manner, but this time the change is recorded within the start-tag of the modified element. For example, the following entry records the change of the value of a single attribute for an element:

```
<date deltaxml:delta="WFmodify"
      deltaxml:old-attributes="datatype='integer'"
      deltaxml:new-attributes="datatype='date'">
```

If a new attribute is added then the `deltaxml:new-attributes` attribute is used on its own. If an attribute is removed then only the `deltaxml:old-attributes` attribute is required to record the change.

Changes to parsed character data (PCDATA) are recorded using `delataxml:PCDATAmodify` elements, as shown in the following example:

```
<p xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
 deltaxml:delta="WFmodify">
 The text
 <deltaxml:PCDATAmodify>
  <deltaxml:PCDATAold>originally shown in</deltaxml:PCDATAold>
  <deltaxml:PCDATAnew>displayed using</deltaxml:PCDATAnew>
 </deltaxml:PCDATAmodify>
 a fixed width font is now bold.
</p>
```

Recording changes to namespaces requires a little more thought. It is important to remember that the part of the qualified name preceding the colon in a namespace qualified element or attribute name is only a shorthand for the URI used to uniquely identify the namespace within the namespace declaration. Before checking whether two files match it is important that all namespace qualifiers be replaced by the relevant URIs. In addition any unqualified names will need to be assigned default namespace URIs if they are nested within an element that specifies a default value. In such cases it must be remembered that the default namespace also applies to the element that declares the default namespace if it does not have a namespace qualifier. Examples 1.1 and 1.2 showed how specific namespaces and default namespaces can interact to produce identical XML information sets from qualified XML names that have different namespace qualifiers.

1.5 The benefits of exchanging delta files

From the simple examples provided so far the you might be misled into thinking that delta files are typically much larger than the source files. But in a typical application this is unlikely to be true.

During most updating sessions only a small proportion of a file will be updated. Unless the verbose format of the DeltaXML needs to be generated for the purposes of printing out the text with the revisions marked in it, for the majority of elements in the file a sequence consisting of a start-tag for the highest unchanged element in the hierarchy, some parsed character data or elements, and an end-tag will be replaced within the delta file produced by DeltaXML by an empty tag with a single attribute whose name and value is a fixed 27 character string. Providing the data, including any unchanged nested elements, plus the end-tag is greater than 27 characters long the length of the uncompressed text file will be reduced. But the minute the file is compressed all of the repetitive attribute values added by DeltaXML will become single tokens, typically resulting in four or five bytes of storage or transmission space. The following examples illustrate the types of savings that can be made:

***Example 0-15* Source with nested elements**

```
<p>Possible applications of DeltaXML include:
  <ul>
    <li>verifying changes to <em>medical records</em></li>
    <li>synchronizing sets of <em>clinical trial data</em></li>
    <li>validating <em>software updates</em></li>
    <li>rolling back <em>editing operations</em></li>
    <li>exporting minimized <em>update files</em>.</li>
  </ul>
</p>
<p>Many other applications of this versatile suite of Java modules
have been developed, in industries as diverse as
telecommunications, pharmaceutical, healthcare, legal
documentation, loose-leaf publishing, software engineering,
banking and insurance.</p>
<p>The uses to which you will be able to make of the DeltaXML
toolkit will depend on the diversity of your local XML
applications. Most of our customers find that, having bought
DeltaXML to help with a specific application, they have many
different uses for the suite.</p>
```

Example 0-16 ***Updates with nested elements***

```
<p>Possible applications of DeltaXML include:
  <ul>
    <li>identifying updates needing <em>translation</em></li>
    <li>verifying changes to <em>medical records</em></li
    <li>synchronizing sets of <em>clinical trial data</em></li>
    <li>validating <em>software updates</em></li>
    <li>exporting minimized <em>update files</em>.</li>
  </ul>
</p>
<p>Many other applications of this versatile suite of Java modules
have been developed, in industries as diverse as
telecommunications, pharmaceutical, healthcare, legal
documentation, loose-leaf publishing, software engineering,
banking and insurance.</p>
<p>The uses to which you will be able to make of the DeltaXML
toolkit will depend on the diversity of your local XML
applications. Most of our customers find that, having bought
DeltaXML to help with a specific application, they have many
different uses for the suite.</p>
```

Example 0-17 ***DeltaXML delta file comparing 1.15 with 1.16***

```
<p deltaxml:delta="WFmodify">
 Possible applications of DeltaXML include:
 <ul deltaxml:delta="WFmodify">
  <li deltaxml:delta="add">
   identifying updates needing
   <em>translation</em>
  </li>
  <li deltaxml:delta="unchanged" />
  <li deltaxml:delta="unchanged" />
  <li deltaxml:delta="unchanged" />
  <li deltaxml:delta="delete">
   rolling back
   <em>editing operations</em>
  </li>
  <li deltaxml:delta="unchanged" />
 </ul>
</p>
<p deltaxml:delta="unchanged" />
<p deltaxml:delta="unchanged" />
```

Recording changes in an XML-encoded delta file means that changes can easily be interchanged with other XML-aware systems using the Simple Object Access Protocol (SOAP). The transmitted files can be merged with locally held copies of the source file from which the delta file has been calculated to produce a file that matches the edited file on the transmitting system.

DeltaXML also provides a three-way merge option. This means that two computers can both transmit the changes made to the last synchronized source file on their local systems and can integrate the changes made locally with those recorded remotely to produce a new synchronized version of the file.

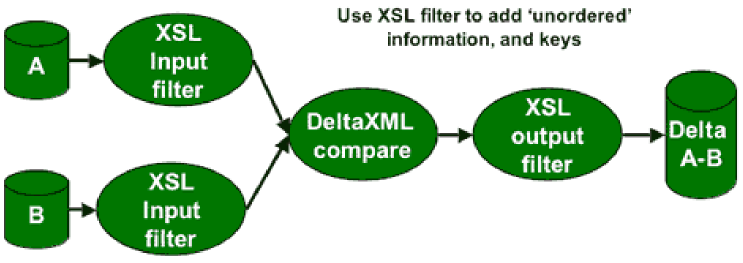
1.6 DeltaXML – Understanding XML changes

DataXML is distributed as a set of Java *factories*. Factories are used to create instances that conform to particular Java *interfaces*. The DeltaXML API follows the SAX standard of using *features* and *properties* to select options during comparisons and recombinations. Properties and features are used to configure either a `RunnableXMLComparator` or a `RunnableXMLCombiner` using `comparator.setProperty`, `comparator.setFeature`, `combiner.setProperty` and `combiner.setFeature` methods. A typical application of the API would take the form:

```
BriefcaseFactory bFactory = BriefcaseFactory.newInstance();
  Briefcase briefcase = bFactory.newBriefcase();
  RunnableXMLComparator comparator = cFactory
  .newRunnableXMLComparator(briefcase, source1, source2,
                            filter3);
  comparator.setFeature("http://deltaxml.com/api/isFullDelta",
                        true);
  comparator.runNow();
```

The API allows calls to the differencing suite to be made directly from an XML parser. It also allows pipelines to be created, for example so that XSL Transformations can be used to pre-process or post-process the XML source and result files. A pipeline for doing word-by-word comparisons of parsed character data is supplied with the API. The API also allows multithreaded applications to be developed.

Where the ordering of elements is not considered important, items to be randomly checked can be assigned a *key* based on either unique identifiers already assigned to the element, or some unique combination of attributes, contents or generated data that can be ensure unique identification of each element. Such keys, which are added to the relevant element in the form of a `deltaxml:key` attribute, are normally generated as the result of an XSL Transformation on the source files. The keys on the elements in two *unordered* source files processed using matching rules can be compared, with the result being passed through a XSL Transformation after comparison to allow a visible representation of the delta file to be created.



The same technique can also be used to identify changes made between different versions of a DTD or XML schema. A transformation is developed that converts one or both of the different models to a matching model. When appropriate transformations have been applied to the source files they can be compared as if one were a simple update of the other.

Keys can also be useful when trying to compare structured text where the differences between the changed items are not clear. If keys are not assigned to differentiate paragraphs of the type shown in Examples 1.18 and 1.19 you end up with reports such as that shown in Example 1.20:

Example 0-18 ***Original without using keys***

```
<p>Para 1a: These paragraphs have no keys.</p>
<p>Para 1b: Therefore it is difficult to detect when a paragraph
has  been changed and when it has been deleted and a new one
added.</p>
<p>Para 1c: This paragraph will be deleted, and a new one added
after the next paragraph.</p>
<p>Para 1d: This paragraph will be modified.</p>
```

Example 0-19 ***Changes without using keys***

```
<p>Para 1a: These paragraphs have no keys.</p>
<p>Para 1b: Therefore it is difficult to detect when a paragraph
has been changed and when it has been deleted and a new one
added.</p>
<p>Para 1d: This paragraph will be modified - like this.</p>
<p>Para 2a: This paragraph has been added in version 2.</p>
```

Example 0-20 ***Formatted DeltaXML result without using keys***

Para 1a: These paragraphs have no keys.

Para 1b: Therefore it is difficult to detect when a paragraph has been changed and when it has been deleted and a new one added.

Para ~~1c:~~1d: This paragraph will be ~~deleted, and a new one added after the next paragraph.~~modified - like this.

Para ~~1d:~~2a: This paragraph ~~will be modified.~~ has been added in version 2.

By assigning keys to individual paragraphs to allow them to be uniquely identified the result can be improved to correctly match unchanged paragraphs, allowing changes to be more accurately reported. In this case the keys used by DeltaXML were the unique identifiers assigned to the paragraphs shown in Examples 1-21 and 1-22. The result of the keyed comparison is shown in Example 1-23.

Example 0-21 ***Original using keys***

```
<p id="p1a">Para 1a: These paragraphs do have keys.</p>
<p id="p1b">Para 1b: Therefore it is easy to detect when a
paragraph has been changed and when it has been deleted and a new
one added.</p>
<p id="p1c">Para 1c: This paragraph will be deleted, and a new one
added after the next paragraph.</p>
<p id="p1d">Para 1d: This paragraph will be modified.</p>
```

Example 0-22 ***Changes using keys***

```
<p id="p1a">Para 1a: These paragraphs do have keys.</p>
<p id="p1b">Para 1b: Therefore it is easy to detect when a
paragraph has been changed and when it has been deleted and a new
one added.</p>
<p id="p1d">Para 1d: This paragraph will be modified - like
this.</p>
<p id="p2a">Para 2a: This paragraph has been added in version
2</p>
```

Example 0.23 ***Formatted DeltaXML result using keys***

Para 1a: These paragraphs do have keys.

Para 1b: Therefore it is difficult to detect when a paragraph has been changed and when it has been deleted and a new one added.

~~Para 1c: This paragraph will be deleted, and a new one added after the next paragraph.~~

Para 1d: This paragraph will be ~~modified.~~ modified - like this.

Para 2a: This paragraph has been added in version 2

By combining the various techniques provided as part of the DeltaXML API with existing techniques for parsing, transforming and displaying XML files users are able to create complex applications that greatly reduce the amount of data they need to compare, transmit, print or update. Because DeltaXML files are themselves interchangeable structured XML files they can be transformed, transmitted and displayed on browsers, as well as being merged with other files using either the DeltaXML 2 and 3-way merge options or normal XSL Transformations.

Using XML to record changes in XML files greatly enhances the number of ways in which change data can be used, while at the same time simplifying the change management process by allowing it to be integrated with the rest of the site's data management operations. As the XML files used to record the changes can also be stored as part of your data repository they can become a permanent, auditable, record of the process of updating structured data sets.