Title:	A comparative study of XML diff tools
Authors:	Grégory Cobéna
	Talel Abdessalem
	Yassine Hinnach
Contact :	Grégory Cobéna
	Gregory.Cobena@inria.fr
	+33 (0)1 39 63 56 62
Topic:	Semi-structured Data, Metadata and XML
Keywords:	XML, version management,
	diff tools, change detection,
	representing changes
Address:	INRIA Futurs - Projet GEMO
	Parc Club Orsay Université
	4 rue Jacques Monod, bat G
	91893 Orsay Cedex
	France

A comparative study of XML diff tools

Grégory Cobéna[†], Talel Abdessalem[‡], Yassine Hinnach[‡]

Abstract

The success of XML has recently renewed interest in change control on trees and semi-structured data. This is motivated, for instance, by the need to manage versions of documents, to query and monitor changes and to efficiently exchange documents and their updates. In many applications, the changes that occurred between two versions of a document are unknown to the system. Hence, a *diff* algorithm is used to construct a *delta* representing the changes.

Various diff algorithms have been proposed. Some run in quadratic time and space cost while others run in linear time. Some consider the tree structure of XML documents while others do not. Also, some algorithms may find a more "concise" sequence of changes. This improves the quality of monitoring and querying changes. We study here different tools and compare them based on experiments conducted over large sets of XML data.

There are also several proposals for delta formats to represent the changes in XML, but there is no standard yet. We study recent proposals based on their specifications, their implementations and on experiments that we conducted.

Our goal is to provide an evaluation of the performance and quality of the tools and, based on this, guide the users in choosing the appropriate solution for their applications.

Keywords

XML, version management, diff tools, change detection, representing changes.

INTRODUCTION

Users are interested in gathering knowledge and data from the Web or other data warehouses. One may remark that users often search for News. Consider for instance a person who is interested in Art, or History. Even if there is already a very large amount of available knowledge on the topic, this person often wishes to subscribe to news magazines, mailing lists or newsletters to be regularly informed. Indeed, users are often interested as much (if not more) in changes of data, e.g. new data, as in the data itself.

[†] PCRI (France), a joint lab between INRIA, Ecole Polytechnique, Université Paris-Sud and CNRS
[‡] ENST Paris, France

Thus, change-control is an important topic that we study here. Typically, documents are collected periodically, for instance by crawling the Web. When a new version of an existing document arrives, we want to understand changes that occurred since the previous version. Considering that we have only the old and the new version of a document, and no information on what happened in between, a *diff* algorithm is used. A typical setting for the diff is as follows: the input consists of two files representing two versions of the same document, the output is a *delta* file representing the changes that occurred between them.

The precise context of the present work is changes in tree-structured data. With the Web and standards such as HTML and XML, tree data is becoming extremely popular. Today, XML [1] is a de facto standard for exchanging data both in academia and in industry. We also believe that XML is becoming a standard to model stored data. Our work was performed in the context of the Xyleme project [2] that studies and builds a dynamic XML warehouse, capable of storing massive volumes of XML data. XML allows for real query languages [3], [4], [5] and facilitates semantic data integration [6], [7].

Several diff algorithms exist for text files, the most famous being *GNU Diff* [8]. It would be possible to consider the text representation of XML documents and apply text diff algorithms to detect changes. But XML documents are more than text. Their tree-structure gives semantics to pieces of information, in particular if an XMLSchema is provided [9]. In the present work, we focus on diff algorithms for XML that take advantage of the tree structure and its associated semantics [10].

There has been a lot of work on the topic of *tree diff* algorithms. However, the case of XML requires some essential modifications of these algorithms. Primarily because in XML, the “data” is mostly stored in leaves (text nodes and attributes values), whereas the labels of internal nodes (elements and attributes names) represent the schema. Also new algorithms have been proposed directly for XML. This motivates the need for an in-depth comparison of XML diff tools. Although there are comparisons for diff on trees [11], no such comparison exists for XML, to our knowledge.

There are two dimensions to the problem: (i) the representation of changes, in order to use the change information, and (ii) the detection of changes.

(i) Representation. The *deltas* we consider here are XML documents describing the changes. The choice of XML is motivated by the need to exchange, store and query these changes. Since XML is a flexible format, there are different possible ways of representing the changes on XML and semi-structured data [12], [13], [14], [15], and building version management architectures [16].

We compare several proposals of delta formats to represent changes. All delta formats represent changes as insert/delete/update operations, some also support move operations. Some formats describe the operations that transform one version into another. Other formats have deltas that are a summary of the document versions, enriched by annotations (e.g. attributes) that indicate the change status for each node. Formats also have various features, such as providing "backward" information to transform the newer version into the older one (i.e. reverse the delta). However, we found out that most formats are "almost" equivalent, where equivalence means the possibility to convert a delta from one format to another without loss of information. In order to verify that claim, we implemented such conversion tools. We also found out that the identification of nodes is a crucial aspect. Identifying nodes is necessary to describe changes, and may also be useful to improve query efficiency (e.g. add structure information), or temporal queries (e.g. persistent identification). We detail some existing proposals and compare them.

(ii) Detection. We consider 3 main classes of algorithms and compare their performance and quality. Note that if *move* operations are considered, the tree diff problem is NP-hard. In absence of move, algorithms in the first class find "minimal" results in quadratic time/space cost, e.g. a few hours for a megabyte XML document. Algorithms in the second class compute deltas close to the minimal with significantly reduced computation time. Finally, algorithms in the third class compute in quasi linear time a reasonable result. The results of all diffs that we consider allow to transform the old version of the document into the newer one. They may differ in conciseness. Minimality is important here not only for efficient version management, but also because smaller deltas are more "concise" and also likely to be closer to "real" changes (i.e. the one that were performed by the user). Finding a "real" or "concise" sequence of changes improves in general the quality of monitoring and querying them.

Experiment Settings. Our comparative study relies on experiments conducted over XML documents found on the web. Xyleme crawler [2] was used to crawl more than five hundred millions web pages (HTML and XML) in order to find five hundred thousand XML documents. Because only part of them changed during the time of the experiment (several months), our measures are based roughly on a hundred thousand XML documents. Only some experiments were run on the entire set. Most were run on sixty thousand documents because of the time it would take to run them on all the available data. It is also interesting to run it on private data (e.g. financial data, press data). Such data is typically more regular. For instance, we ran our tests on several versions of XML data from DBLP [17]. We also conducted experiments using our change simulator [18], as it gave us more flexibility to study various parameters (e.g. change rate, changes on different types of nodes, distribution of different change operations). We intend to conduct more experiments in the future.

Remark. Observe that this work can also be used for HTML documents by XML-izing them first. However, change management (detection+representation) for a "true" XML document is (in general) semantically much more informative than for HTML documents. It may be also be used for change control on XMLSchemas [9] or even DTDs (by transforming them to XMLSchema first).

The rest of the paper is organized as follows. First, we present the motivations for change management on semi-structured data. In Section II, we present the data model, operations model and cost model. Then, we compare change representations in Section III. The next section is a comparative study on change detection algorithms and their implementation. We present experiments in Section V. Section VI recalls some related work. The last section concludes the paper.

I. Motivations

In this section, we present the main motivations for change management.

(i) Representing Changes. To understand the important aspects of changes representation, we point out some possible applications. Consider a news agency, such as AFP or Reuters. Journalists are editing news articles and news wires. These XML documents are stored in some XML data warehouse.

• *Version Management.* In our example, the news articles may be modified several times by different journalists, and several different versions may be sent to various customers. It is necessary to archive all the versions that have been edited and have been sent out. Version management consists of archiving all versions of each XML document and being able to efficiently retrieve past versions. Several strategies have been proposed and studied [16], [19], [14]. For instance, a possible strategy is to store only the latest version of the document, and all backward deltas that enable to reconstruct past versions [14]. However, the cost of reconstructing past versions increases as the document is modified times and times again. Another possible strategy may be to store all versions of the documents, but it increases the storage cost. In the present paper, we focus on change representation of a delta, i.e. from one version to the another. The change representation should allow for effective *storage strategies*, such as adaptive strategies [19], and efficient *reconstruction of versions* of the documents.

• *Temporal Queries.* One may want to query not only the current value of data, but also past values. For instance, a user may want to retrieve the previous title and first paragraph of all news wires which title has been modified in the last few days. Users may also want to process change information. Consider for instance stock-market information, users may issue queries which involve past values (e.g. the stock prices) and their changes. In all these cases, it is necessary for the system to be able to find and retrieve previous versions of precise nodes from the XML document. The deltas represent changes, but the system may also need to retrieve parts of the document that did not change. Thus, a *persistent identification* of XML tree nodes may be added to the delta in order to improve the efficiency of tracing a node through time. Models for querying multiple versions are presented in [20] and [21].

• *Monitoring and Surveillance.* In some applications, users issue continuous queries to be notified when specific events arise. The spectrum of possible applications for monitoring XML data is very broad (e.g. sensors, web surveillance) [22], [23]. The trigger mechanism involves *queries on changes* that need to be executed in real-time. For instance a journalist may want to be notified each time someone modifies the title of a news wire of which he is an author. Monitoring such events requires to monitor the deltas in order to find out the corresponding change events (e.g. document title has changed), and also to process the original document (e.g. finding who is the author of the corresponding document). Also some changes can only be interpreted in the context of the original document.

To summarize, processing changes requires (i) efficient management of past versions of each node, (ii) efficient integration of change information with the original document.

(ii) Change Detection. In some applications, the system may know exactly which changes have been made to a document. For instance, the graphical interface used by journalists to edit articles may store all editing operations that modify the documents[1]. However, in many cases, the sequence of changes that occurred between two versions of a document is unknown. Thus, the most critical component of change control is the *diff* module that detects changes between an old version of a document and the new version. The input of a *diff* program consists of these two documents, and possibly their DTD or XMLSchema. Its output is a *delta* document representing the changes between the two input documents. Important aspects are as follows:

- *Completeness:* A diff is *complete* if it finds a set of operations that is sufficient to transform the old version into the new version of the XML document. In other words, a diff is complete if it misses no changes. For some application, one may want to trade completeness for performance, for compactness of the delta or to focus only on certain changes. This is not considered here, all diff algorithms we present here are complete.
- *Minimality:* In some applications, the focus is on the minimality of the result (e.g. number of operations, edit cost, file size) generated by the *diff*. This notion is explained in Section II. Minimality of the result is important to save storage space and network bandwidth. Moreover, the efficiency of version management depends on the size (and edit cost) of deltas since smaller deltas result in shorter computation time and lower cost to update the stored data. Minimality is important to obtain concise deltas. Indeed, a more concise sequence of changes is (in general) better than a longer one if they both lead to the same result. The intuition is that longer sequences of changes do unnecessary operations, so they have weaker semantics. In other words, concise deltas are likely to be closer to the "real changes" that occurred. During our experiments, some (limited) human checking of deltas confirmed that minimality improves their quality and semantics.

[1] For instance Microsoft Word stores recent edit operations

• *Performance and Complexity:* With dynamic services and/or large amounts of data, good performance and low memory usage become mandatory. In the context of XML change detection, there is a large range of possible performance and complexity for the algorithms. On one hand some statements of the general change-detection problem are NP-hard, on the other hand, some algorithms have been proposed that run in linear time in the size of the data.

• *"Move" Operations:* The capability to detect 'move' operations (see Section II) is only present in certain *diff* algorithms. The reason is that it has an impact on the complexity (and performance) of the *diff* and also on the minimality and the semantics of the result.

• *Semantics:* In this paper, we consider algorithms that work on the tree structure of XML documents. So they do not only diff the data itself, they also provide results with more precise semantics. For instance, using the tree structure, the tree diff shows that the title of a news wire has changed, because the text update occurs in a node which is child of the `title` node. Text diff programs do not consider the tree structure of XML. Some algorithms may also consider more semantics than just the tree structure of XML documents. For instance, they may consider keys (e.g. ID attributes defined in the DTD) and match with priority two elements with the same tag if they have the same key. Consider a news wire about stock values. The document contains a list of symbols and their last trade value.

```
Yesterday: <trade symbol="MSFT" value="$100" />
           <trade symbol="IBM"  value=" $90" />
           <trade symbol="YHOO" value=" $95" />
           ...
Today    : <trade symbol="MSFT" value="$105" />
           <trade symbol="IBM"  value=" $95" />
           <trade symbol="YHOO" value="$110" />
           ...
```

To correctly understand changes, the algorithm should know that the text below `symbol` is a *key* for each `<trade>` node. Thus, it can show for each company (identified by its symbol) the changing value of the last trades. In this example, `MSFT` increases from 100 to 105. On the other hand, a tree diff algorithm that ignores keys may be confused by similar *values* for different *trade* nodes. As a result, it may detect changes that consist of updating the symbols instead of updating the trade values. For instance: update the `trade` node with `value=95` by changing its `symbol` from 'YHOO' to 'IBM'. One would like to disallow such changes that do not correspond to real changes.

II. Preliminaries

In this section, we introduce the notions that are used along the paper. We use ordered labeled trees as data model for XML documents (DOM [10]). We also mention some algorithms that support unordered trees.

Operations. The change model is based on editing operations as in [14], namely *insert, delete, update* and *move*. There are two possible interpretations for these operations: Kuo-Chung Tai's model [24] and Selkow's model [25], as shown in Figure 1

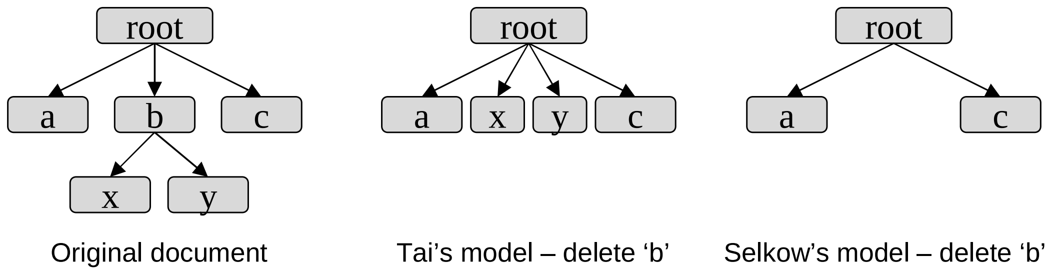

Fig. 1. Tree-Edit models

In Tai's model [24], deleting a node means making its children become children of the node's parent. This model may not be appropriate for some XML documents, since deleting a node may invalidate the document structure according to its DTD (or XMLSchema). In general, this model is not appropriate for object models where the type of objects and relations between them is important [10]. It seems more appropriate, for instance, to applications such as biology where XML is used to represent DNA sequences [26].

Thus, most XML diff tools use Selkow's model [25] in which operations are only applied to leaves or subtrees. In particular, when a node is deleted, the entire subtree rooted at the node is deleted. This captures the XML semantic better, for instance removing a product from a catalog by deleting the corresponding subtree. Other important aspects presented in [14] include (i) management of positions in XML documents (e.g. the position of sibling nodes changes when some are deleted), and (ii) consistency of the sequence of operations depending on their order (e.g. a node can not be updated after one of its ancestors has been deleted).

Edit Cost. The *edit cost* of a sequence of edit operations is defined by assigning a cost to each operation. Usually, this cost is 1 per node touched (inserted, deleted, updated or moved). If a subtree with n nodes is deleted (or inserted), for instance using a single delete operation applied to the subtree root, then the edit cost for this operation is n. Since most *diff* algorithms are based on this cost model, we use it in this study. The *edit distance* between document A and document B is defined by the minimal edit cost over all edit sequences transforming A in B. A *delta* is *minimal* if its edit cost is no more than the edit distance between the two documents.

One may want to consider different cost models. However, some cost models imply trivial solutions to the diff problem. Consider for instance, the case of assigning $cost = 1$ to each edit operation, e.g. deleting or inserting an entire subtree. When two documents are different, a minimal edit script would often consist of the following pair of operations: (i) delete the first document with a single 'delete' operation applied to the document's root (ii) insert the second document with a single 'insert' operation.

The 'move' operation. The semantics of 'move' is to identify nodes (or subtrees) even when their context (e.g. ancestor nodes) has changed. Some of the proposed algorithms are able to detect move operations between two documents, whereas others are not. The reason is that most formulations of the change detection problem with 'move' operations are NP-hard [27]. So the drawback of detecting moves is that the algorithms that can be used in reasonable time only approximate the minimum edit script. In [27], the authors consider the problem of comparing two CUAL (Connected, Undirected, Acyclic and Labeled) graphs. By reduction from exact cover by 3-sets, they show that finding the distance between two graphs is NP-hard. They extend this by proposing a constrained distance metric, called the degree-2-distance, requiring that any node to be inserted (deleted) has no more than 2 neighbors. In this view, diff algorithms based on Selkow's model correspond to finding the degree-1 distance.

The improvement when using a move operation is that in some applications, users may consider that a move operation is more intuitive (or less costly) than a deletion and insertion of the same subtree. It often corresponds to reality, e.g. in a storage file-system, moving a directory of files is cheaper than copying (and then deleting) them, and is also cheaper than moving each file one-by-one. Thus, the cost of a move operation is '1'. In temporal databases, move operations are important to detect from a semantic viewpoint because they allow to identify (i.e. trace) nodes through time better than delete and insert operations.

Mapping/Matching. In the next sections, we also use the notion of "mapping" between two trees. Each node in tree A (or tree B) that is not deleted (or inserted) is "matched" to its corresponding node in B (or A). A *mapping* between two documents represents all matchings between nodes from the first and second documents. The notion of "minimal" delta can be defined with respects to a given mapping between the two documents. Some algorithms (e.g. LaDiff, see below) work like this.

The definition of the mapping and creation of a corresponding edit sequence are part of the "change detection". The "change representation" consists of a data model for representing the edit sequence.

III. Change Representation models

We have seen that an important aspect of version management is the representation of changes, but a standard is still missing. In this section, we present the four recent proposals to represent changes of XML documents in XML using deltas. A typical delta describes the operations that transform one version of the document into another, e.g. by inserting/deleting nodes. We selected these proposals because they seem to be the most representative of the various possibilities, and because they have been fully implemented. We also mention briefly some other proposals that we studied.

This section is organized as follows. We first present three proposals for identifying nodes. They are used by the delta formats. Then, we present the four delta formats. Finally, we give a summary of the different formats, their features and the equivalence between them.

Remark. One may want to use the flat text representation of changes, e.g. which can be obtained with the GNU diff tool. This representation is indeed efficient to manage the text versions of XML data. However, many XML warehouses use object-relational or native representations of XML data [28], and the text deltas can not be used in this context. Also, more work would be necessary to enable temporal queries and monitoring changes with text deltas.

A. *Identifying nodes*

Identifying nodes is an extremely important aspect in managing XML data. In Figure 2, we present an example of an XML document, and 3 formats for identifying its nodes. These are XPath, XDL and XID.

XPath [4] is the current standard (from the W3C) for retrieving parts of a document. Its model for identifying nodes is used in XQuery [3], the standard query language for XML data. In our context, the three drawbacks of XPath are:

- (D1) A given XPath expression can not (in general) be used to identify the same node in different versions of the document.
- (D2) An XPath expression does not (in general) return exactly one node. It returns a set of node, which may be empty or contain one or several nodes.
- (D3) Several different XPath expressions can be used to identify a given node.

XDL-Path, which is in the same spirit of XPath, has been proposed in the context of the Microsoft XmlDiff tool [29]. XDL could be seen as a subset of XPath, since it consists of a specific XPath expression that uses only the *child* axis and the *position* information. This fixes the two drawbacks (D2) and (D3), but still does not fix (D1).

XIDs are used in the context of *XyDelta* [14]. The XID identification proposal consists of assigning a unique and persistent label to each node. The label is unique in that in each version of a document, there is no more than one node for each label. It is persistent in that it identifies the versions of a node through time. In our example, the first `product` node is identified by XID 7. Thus, in Figure 3, the corresponding `product` node is assigned the same XID. This proposals fixes (D1), (D2) and (D3). But its drawback is that the assignment of a node to an XID (and an XID to a node) has to be maintained by the system. To do so, the notion of *XID-map* is proposed in [14]. It is a compressed format that stores, for a given version of a document, a list of persistent identifiers. In this list, there is exactly one XID for each node of the XML tree.

Assigning labels to nodes may be useful for improving query performance. For instance, labeling nodes with both their prefix and postfix position in the tree allows to quickly compute ancestor/descendant tests and thus significantly improves querying [5]. But prefix/postfix labels are not persistent. In principle, it would be nice to have one labeling scheme that contains both structure and persistence information. However, [30] shows that this requires longer labels and uses more space. In [31], the authors address path-expression queries on multi-version XML documents, and they use a numbering scheme, in the spirit of XIDs, which includes both structural and persistency information.

```
                          | XID | XDL Path | XPath
<catalog>                 |  15 | .        | .
  <product>               |   7 | /1       | /product[1]
    <name>                |   2 | /1/1     | /product[1]/name
      Notebook            |   1 | /1/1/1   | /product[1]/name/text()
    </name>               |     |          |
    <description>         |   4 | /1/2     | /product[1]/description
      2200MHz Pentium4    |   3 | /1/2/1   | /product[1]/description/text()
    </description>        |     |          |
    <price>               |   6 | /1/3     | /product[1]/price
      $1999               |   5 | /1/3/1   | /product[1]/price/text()
    </price>              |     |          |
  </product>              |     |          |
  <product>               |  14 | /2       | /product[2]
    <name>                |   9 | /2/1     | /product[2]/name
      Digital Camera      |   8 | /2/1/1   | /product[2]/name/text()
    </name>               |     |          |
    <description>         |  11 | /2/2     | /product[2]/description
      Fuji FinePix 2600Z  |  10 | /2/2/1   | /product[2]/description/text()
    </description>        |     |          |
    <status>              |  13 | /2/3     | /product[2]/status
      Not Available       |  12 | /2/3/1   | /product[2]/status/text()
    </status>             |     |          |
  </product>              |     |          |
</catalog>                |     |          |

(Note: Node identifiers based on XID, XDL-Path and XPath are given)
```

Fig. 2. First version of a document

```
<catalog>
  <product>
    <name>Notebook</name>
    <description>2200MHz Pentium4</description>
    <price>$1999</price>
  </product>
  <product>
    <name>Digital Camera</name>
    <description>Fuji FinePix 2600Z</description>
    <price>$299</price>
  </product>
</catalog>
```

Fig. 3. Second version of the document

B. Edit operations

We now present the four change representation models: XUpdate, XDL, XyDelta and DeltaXML.

XUpdate. XUpdate is a proposed standard by XML DB [15], it is designed to provide means to update XML data. It describes edit operations, insert/delete/update, based on Selkow's model. The nodes affected by the operations are identified in the original document using XPath expressions. The delta corresponding to Figures 2 and 3 is:

```
<xupdate:modifications version="1.0"
 xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/catalog[1]/product[2]/description[1]" >
    <xupdate:element name="price">
      $299
    </xupdate:element>
  </xupdate:insert-after>
  <xupdate:remove select="/catalog[1]/product[2]/status[1]" />
</xupdate:modifications>
```

The drawbacks are as follows:

- It does not support more complex operations, such as 'move'
- It can not be reversed, i.e. given the delta from $V1$ to $V2$ we can not construct the delta from $V2$ to $V1$.

Reversing a delta consists intuitively in replacing insert operations by delete, and delete operations by insert. Here, the content of nodes that are deleted is not stored in the delta, thus the reversed delta does not know which content should be inserted.

An advantage of using XPath expressions is that they give an intuition of the context in which the changes are applied, e.g. the tags of ancestor nodes. This may be used to implement simple monitoring features on the delta that do no require reading the original document, e.g. find all operations which change the `price` of a `product`. Finally, XUpdate seems to support the use of *Variables* , but a more precise proposal is clearly needed for such an advanced feature.

XyDelta. XyDelta [14] is a format that was designed in the context of the Xyleme project [2]. It describes edit operations according to Selkow's model. The nodes affected by the operations are identified using XIDs, which are persistent identifiers that we presented previously. Each *move* operation is represented by a pair of *insert* and *delete* operations that apply to the same node (i.e. same XID). Let us explain the example in

Figures 2 and 3. Nodes 12-13 (i.e. from 12 to 13) that have been deleted are removed from the *XidMap* of the second version, while new identifiers (e.g. 16-17) are assigned to inserted nodes. The corresponding delta is:

```
<xydelta
  v1_XidMap="(1-15)"
  v2_XidMap="(1-11;16-17;14-15)">
  <delete xid="(12-13)" parent="14" position="3">
    <status>Not Available</status>
  </delete>
  <insert xid="(16-17)" parent="14" position="3">
    <price>$299</price>
  </insert>
</xydelta>
```

XyDeltas have mathematical properties. They can be aggregated, i.e. given a XyDelta from $V1$ to $V2$, and a XyDelta from $V2$ to $V3$, it is possible to construct the XyDelta from $V1$ to $V3$. They can also be reversed, i.e. construct the delta from $V2$ to $V1$. Also the formal model of XyDelta considers each XyDelta having a *set* of operations, instead of a *sequence* of operations [32]. This facilitates comparisons of deltas, synchronization of deltas and applying only parts of a delta to some document. The main drawback of XyDeltas is that they require to explicitly maintain the persistent identifiers. Moreover, XIDs do not give the intuition about the context in which changes occur. To understand the context in which changes occur (e.g. ancestor nodes), the XIDs are used to retrieve the corresponding nodes from the original document.

Microsoft XDL. XML Diff Language (XDL) [29] is a proprietary XML-based language for describing differences between two XML documents. An instance of XDL is called the *XDL diffgram*. This language is the most recent proposal. The difference with previous work is that it is based on Tai's edit model for trees. In a way similar to XyDelta, *move* operations are represented as a pair of insert/delete operation. The nodes affected by *delete* or *update* operations are identified by a `match` attribute. It contains a path (see previous XDL-Path), which is relative to the current "context node", and which designates the "context node" for child operations. For instance, a *delete* operation can be described as follows:

```
<xd:delete match="/2/3" />
```

It is also possible to use an `xd:node match` operation in order to define a new *context* node. For instance, previous deletion would be as follows:

```
<xd:node match="2">
  <xd:delete match="3" />
```

```
</xd:node>
```

This is useful for *insert* operations, since the target position is given by the current context node.

The drawback of XDL compared to XyDelta is that the diffgrams can not be reversed or aggregated. The drawback compared to XUpdate is that the path descriptor used in XDL is not sufficient to implement simple monitoring features. XDL offers several advantages: (i) the verification of source and target documents using a hash value that is stored in the delta, (ii) a *copy* operation which may save space in some cases, and (iii) XDL is validated by a DTD.

The fact that XDL uses Tai's model instead of Selkow's model may be seen are an advantage or a drawback depending on the application requirements. For instance, this model includes more operations than Selkow's model such as renaming an element node in the XML document. This results in smaller deltas in some cases, and may also be useful in applications were these updates make sense. On the other hand, as described previously, these operations may be not so well adapted to some DTDs and XML schemas.

A typical example of the diffgram (delta) corresponding to Figures 2 and 3 is as follows. Using Tai's model, removing the `status` and inserting the `price` can be seen as replacing the tag `status` by `price` (see the `xd:change` operation). The text node `"Not Available"` is updated to `"$299"`.

```
<xd:xmldiff
  srcDocHash="......"
  xmlns:xd="http://schemas.microsoft.com/xmltools/2002/xmldiff" >
  <xd:node match="1">
    <xd:node match="2">
      <xd:change match="3" name="price">
        <xd:change match="1">$299
        </xd:change>
      </xd:change>
    </xd:node>
  </xd:node>
</xd:xmldiff>
```

DeltaXML is a format that has been presented in [13], and is in the spirit of [33]. The delta consists of a ”summary” of the original document, to which attributes are added to describe the ”change” operations such as delete, insert and update. The identification of nodes with the same parent and same element name is achieved by also storing the sibling nodes that did not change, annotated with a `deltaxml:unchanged` attribute. In other words, there is a trivial mapping between the delta and the original document, and the delta has the same look'n'feel as the original document. Note that *move* operations are not supported. Change attributes use the `deltaxml` namespace. The delta corresponding to Figures 2 and 3 is:

```
<catalog
    xmlns:deltaxml="http://www.deltaxml.com/ns/well-formed-delta-v1"
    deltaxml:delta="WFmodify" >
    <product deltaxml:delta="unchanged"/>
    <product deltaxml:delta="WFmodify">
        <name deltaxml:delta="unchanged"/>
        <description deltaxml:delta="unchanged"/>
        <deltaxml:exchange>
            <deltaxml:old>
                <status deltaxml:delta="delete">Not Available</status>
            </deltaxml:old>
            <deltaxml:new>
                <price deltaxml:delta="add">$299</price>
            </deltaxml:new>
        </deltaxml:exchange>
    </product>
</catalog>
```

There are two reasons why the delta is "almost" but not stricly validated by the DTD of the document.

- The delta may not contain the content of subtrees that did not change, e.g. the first product in our example. Depending on the DTD, the validation may require all content to be present. To solve this issue, DeltaXML offers the possibility to also store unchanged parts.
- It is difficult to annotate the text nodes which are updated, for instance text nodes can not have attributes. Therefore, an additional node is used to describe some operations on text nodes. This node is (in general) not supported by the original DTD of the document.

The DeltaXML contains lots of information. This may be useful to implement simple monitoring features as well as some simple queries that would not require processing the original document. The drawback is some storage overhead, in particular when the change rate is low. The two reasons are (i) that the position management is achieved by storing the root of unchanged subtrees, (ii) that the change status is propagated to ancestor nodes, increasing the size of data.

Other Languages. We studied other languages, but they are not really different from the ones that we presented.

A slightly different model is for instance **Dommitt** [34]. This representation of changes is in the spirit of *DeltaXML*. However, instead of using change attributes, new node types are created. For instance, when a `book` node is deleted, a `xmlDiffDeletebook` node is used. Thus, for each document DTD, a new DTD is created which describes the language of deltas for this type of documents. It is nice to have a DTD that describes the language of changes, but the drawback is to have many specific DTDs with specific tag names that also differ from the original document ones.

Text formats. Some tools, such as *GNU diff*, use text formats. For instance, DecisionSoft proposes an Open Source xml diff program [35] using a "text" format as follows. The XML document is displayed in its text format and lines that have been deleted or inserted are annotated.

```
           <author>
DELETE       <name>Stefan Hellkvist</name>
DELETE     </author>
DELETE     <author>
             <name>Magnus Ljung</name>
DELETE       <phone/>
           </author>
```

The insert/delete operations do not apply to the tree structure of XML. For instance, it is possible to delete a `</author>` line. Such operations may be translated into tree operations, but more work would be necessary. In our example, the two `author` subtrees are merged by the deletion of `</author>` and `<author>`.

Text formats enable storage/compression of versions but more work is necessary to query changes or apply the delta on a DOM (tree) representation of the document.

C. Summary

During this study, we tested a few other change formats that are not mentioned here. Some are comparable to those presented here, others turned out to be too limited to really support the tree structure of XML. We summarize next the important aspects of the change formats.

• *Identifying nodes.* As we have seen, there is a trade-off between short node identifiers (e.g. XDL, XID/XyDelta), and longer expressions and/or context nodes which allow to implement simple monitoring features (e.g. DeltaXML, XUpdate). Using persistent identifiers, such as XID (XyDelta), requires the system to maintain this knowledge, but improves significantly the support of temporal queries. Note that persistent identifiers could be stored in addition to other identifiers.

• *Edit Operations.* Some formats use Selkow's model, others (e.g. XDL) use Tai's model. Since Selkow's operations are a subset of Tai's operations, any sequence of operation from DeltaXML, XUpdate or XyDelta can be transformed easily into an XDL delta. The converse is also possible but more difficult. Tai's specific operations can be expressed by a sequence of operations in Selkow's model, but the result obtained is often far more complex to read than the original sequence. For instance, Tai's delete of a node n can be expressed in Selkow's model: move all the child subtrees of n to the parent of n, and then delete n. We implemented such conversion tools, the results are briefly described in Section V. With languages that support move operations (e.g. XyDelta), no information is lost since the persistent identification of nodes is maintained. With other formats (i.e. no move), the persistent identification of nodes is lost. Thus, XDL (and also XyDelta due to 'move' support) has an advantage in terms of expressive power, although we have seen that using Tai's model to modify XML documents does not always improve the semantics of deltas. XyDelta has another advantage since it contains information to reverse deltas.

• *'Move' Operation.* Two formats (XDL and XyDelta) support the *move* operation, whereas DeltaXML and XUpdate do not. We believe that DeltaXML and XUpdate could be extended to support move operations. The move operation is useful for many reasons, for instance because it is a "cheap" way to move a large part of the document. Also note that using move operations is often important to maintain persistent identifiers since using delete and insert does not lead to a persistent identification.

• *Other Features.* The formats presented here have different features, such as validation by a DTD or XMLSchema, using hash values to check documents integrity (e.g. XDL), supporting backward deltas (e.g. XyDelta). It seems that these features could be relatively easily added to all formats that we presented.

• *Multiple deltas.* Only XyDelta offers support to manipulate multiple deltas, providing for instance functions to reverse and aggregate them. Thus, XyDelta has an important advantage here. The other tools may be extended to support such functions, but this requires more work. Also it would be interesting to see how the various delta formats could be used in the context of synchronizing changes (e.g. multiple deltas that apply to the same version of the document). Again, XyDelta seems to have a slight advantage since its model considers independant operations, but more work is clearly needed in this direction.

We believe that one of the most important aspects of change management is missing for all these formats: queries. Each format may be used to construct a monitoring system and/or a temporal query engine, but we are missing a specification and precise framework for using these formats to support such a system.

IV. Change Detection algorithms and tools

In this section, we present an overview of the abundant previous work in this domain. The algorithms and tools we describe are summarized in Figure 5 (page 29). A *diff* algorithm has two parts: first it matches nodes between the two (versions of the same) document(s). Second, it generates a document, namely a *delta*, representing a sequence of changes compatible with the matching. Intuitively, there are four possible sorts of algorithms. When possible, we classified known algorithms according to these categories. The algorithms and tools are presented in detail in next part. The classification is as follows:

1. **(C1)** Algorithms that compute the minimum edit sequence among all possible edit scripts, including scripts with *move* operations. The problem is NP-hard, so these algorithms would have an exponential cost in the size of the data. Actually, we did not find any tool (or publication) that uses such an algorithm.
2. **(C2)** Algorithms that compute the minimum edit sequence among scripts with insert/delete/update operations (no move). The cost (memory and speed) is quadratic in the size of the data. Thus, we found out during our experiments that these algorithms are limited to files up to roughly 500kb on a regular PC, since it takes 1Gb of memory and several hours of computation. In this category, we tested MMDiff, XMDiff, XDiff. XMDiff is an external-memory version of MMDiff, thus the memory cost is constant but replaced by a quadratic I/O cost.
3. **(C3)** Algorithms that are based on quadratic computations of the minimum edit sequence, but with significant changes to run faster. They do not always find the minimum edit script. Some run on average in quadratic time (in the size of the data), others run on average in linear time. However, the worst-case bound remains quadratic, i.e. the computation/memory cost for some files is close to the quadratic cost. In this category, we tested DeltaXML, Microsoft Precise-XmlDiff, Logilab-XmlDiff (Zhang Shasha version).

4. **(C4)** Algorithms that run in linear time. The main idea is to match nodes in both documents, and afterwards generate the edit script that describe corresponding changes. Several techniques may be used to match nodes. One of them is to use the XML structure as much as possible, i.e. propagate matching from node to node if possible (Microsoft Treewalk-XmlDiff, XyDiff). Others rely on ’pattern matching’ techniques tailored to XML documents (XyDiff), or on the content analysis of documents (e.g. finding similar words as in LaDiff). These algorithms are able to find good results (in terms of minimality and semantics) at low cost. The cost for files up to several megabytes is no more than the document’s size in memory, and no more than a few seconds or minutes in time. In this category, we tested XyDiff, Microsoft Treewalk-XmlDiff, Logilab-XmlDiff (Fast version).

For simplicity, we name the categories as follows: (C2) ’quadratic’ algorithms, (C3) ’fast’ algorithms, (C4) ’linear’ algorithms. The goal of our survey is to compare both the performance and the quality of several XML diff tools. In the next sections, we also present experiments on the performance (see Section V-B) and the quality (see Section V-C) of the tools. In this section, we compare the tools based on the formal description of their algorithms (if available), and in particular we consider the upper-bound complexity and the minimality of the *delta* results.

The next subsections are organized as follows. First, we introduce the String Edit Problem. The String Edit Problem is presented with details since most tree-edit approaches are similar in spirit. Then, we consider optimal tree pattern matching algorithms (C2 and C3) that rely on the string edit problem to find the best matching. Finally, we consider approaches (C4) that rely on finding a good mapping between the two trees.

A. Introduction: The String Edit Problem

Longest Common Subsequence (LCS). In a standard way, the *diff* tries to find a minimum *edit script* between two strings. It is based on edit distances and the string edit problem [36], [37], [38], [39]. Insertion and deletion correspond to inserting and deleting a (single) symbol in a string. A cost (e.g. 1) is assigned to each operation. The string edit problem corresponds to finding an edit script of minimum cost that transforms a string x into a string y. A solution is obtained by considering the cost for transforming prefix substrings of x

(up to the i-th symbol) into prefix subtrings of y (up to the j-th symbol). On a matrix $[1..|x|] * [1..|y|]$, a directed acyclic graph (DAG) representing all operations and their edit cost is constructed. Each path ending on (i, j) represents an edit script to transform $x[1..i]$ into $y[1..j]$. The minimum edit cost $cost(x[1..i] \rightarrow y[1..j])$ is then given by the minimal cost of these three possibilities:

$$cost(deleteCharSymbol(x[i])) + cost(x[1..i-1] \rightarrow y[1..j])$$
$$cost(insertCharSymbol(y[j])) + cost(x[1..i] \rightarrow y[1..j-1])$$
$$cost(updateCharSymbol(x[i], y[j])) + cost(x[1..i-1] \rightarrow y[1..j-1])$$

Note that for example the cost for $updateCharSymbol(x[i], y[j])$ is zero when the two symbols are identical. The edit distance between x and y is given by $cost(x[1..|x|] \rightarrow y[1..|y|])$, and the minimum edit script by the corresponding path.

The sequence of nodes that are not modified by the edit script (nodes on diagonal edges of the path) is a common subsequence of x and y. Thus, finding the minimal delta is equivalant to finding the "Longest Common Subsequence" (LCS) between x and y. Note that each node in the common subsequence defines a matching pair between the two corresponding symbols in x and y.

The space and time complexity are $O(|x| * |y|)$. This algorithm has been improved by Masek and Paterson using the "four-russians" technique [40] in $O(|x| * |y|/log|x|)$ and $O(|x| * |y| * log(log|x|)/log|x|)$ worst-case running time for finite and arbitrary alphabet sets respectively.

D-Band Algorithms. In [41], E.W. Myers exhibited a $O(|x| * D)$ algorithm, where D is the size of the minimum edit script. Such algorithms, namely *D-Band* algorithms, consist of computing cost values only close to the diagonal of the matrix. A diagonal k is defined by (i, j) pairs with the same difference $i - j = k$, e.g. for $k = 0$ the diagonal contains $(0, 0), (1, 1), (2, 2), ...$. When using the usual "1 per node" cost model, diagonal areas of the matrix (e.g. all diagonals from $-K$ to K) contain all edit scripts of cost lower than a given value K. Obviously, if a valid edit script of cost lower than K is found to be minimum inside the diagonal area, then it must be the minimum edit script. When k is zero, the area consists solely of the diagonal starting at $(0, 0)$. By increasing k, it is then possible to find the minimum edit script in $O(max(|x| + |y|) * D)$

time. In other words, D-Band algorithms have a linear computation/memory cost when there are few changes (D is small compared to $|x|$ and $|y|$), and a quadratic cost in the worst case (lots of changes, D is close to $max(|x|, |y|)$). Using a more precise analysis of the number of deletions, [42] improves significantly this algorithm performance when the two documents lengths differ substantially. This *D-Band* technique is used by the famous **GNU diff** [8] program for text files.

B. Optimal Tree Pattern Matching

Serialized XML documents can be considered as strings, and thus we could use a "string edit" algorithm to detect changes. This may be used as a raw storage and raw version management, and can indeed be implemented using *GNU diff* that only supports flat text files. However, in order to support better services, it is preferable to consider specific algorithms for tree data that we describe next. The complexity we mention for each algorithm is relative to the total number of nodes in both documents. Note that the number of nodes is linear in the document's file size.

Previous Tree Models (Zhang-Shasha). Kuo-Chung Tai [24] gave a definition of the edit distance between ordered labeled trees and the first non-exponential algorithm to compute it. Considering two documents $D1$ and $D2$, the time and space complexity is quasi-quadratic: $O(|D1| * |D2| * d(D1)^2 * d(D2)^2)$, where $d(D1)$ and $d(D2)$ represent the depth of the two trees. Zhang and Shasha [43], [44] proposed an algorithm with similar methods. The main difference is that it runs in a postorder traversal of the tree (child nodes are visited first, instead of preorder where parent nodes are visited first). The time complexity is $O(|D1| * |D2| * d(D1) * d(D2))$ and the space complexity is $O(|D1| * |D2|)$. This algorithm is used by *Logilab XML Diff* and *Microsoft XML Diff* that we present next. Yang's [45] algorithm is in the same spirit, it finds the syntactic differences between two programs. In Selkow's variant [25], which is closer to XML, the LCS algorithm described previously is used on trees in a recursive algorithm. Considering two documents $D1$ and $D2$, the time complexity is $O(|D1| * |D2|)$.

MMDiff and XMDiff. In [46], S. Chawathe presents an external memory algorithm *XMDiff* (based on main memory version MMDiff) for ordered trees, and according to Selkow's variant. Intuitively, the algorithm constructs a matrix in the spirit of the “string edit problem”, but some edges are removed to enforce that deleting (resp. inserting) a node will delete (resp. insert) the subtree rooted at this node. More precisely, (i) diagonal edges exist if and only if corresponding nodes have the same depth in the tree (ii) horizontal (resp. vertical) edges from (x, y) to $(x + 1, y)$ exists unless the depth of node with prefix label $x + 1$ in $D1$ is lower than the depth of node $y + 1$ in $D2$. For MMDiff, the CPU and memory costs are quadratic $O(|D1| * |D2|)$. With *XMDiff*, memory usage is reduced but IO costs become quadratic.

Unordered Trees (XDiff). In XML, we sometimes want to consider the tree as unordered. The general problem becomes NP-hard [47], but by constraining the possible mappings between the two documents, K. Zhang [48] proposed an algorithm in quasi quadratic time. In the same spirit is **X-Diff** [49] from the project NiagaraCQ [22]. In these algorithms, for each pair of nodes from $D1$ and $D2$ (e.g. the root nodes), the distance between their respective subtrees is obtained by finding the minimum-cost mapping for matching children (by reduction to the minimum cost maximum flow problem [48], [49]). More precisely, the complexity is $O(|D1| * |D2| * (deg(D1) + deg(D2)) * log(deg(D1) + deg(D2)))$, where $deg(D)$ is the maximum outdegree (number of child nodes) of D. We did not experiment on unordered XML trees. However, we expect that the characteristics of these algorithms are similar to MMDiff and both find a minimum edit script in quadratic time. This is confirmed by experiments presented in [49].

DeltaXML. DeltaXML [50] is the established market leader. It uses a similar technique based on longest common subsequence computations. It uses Wu [42], [41] *D-Band* algorithm to run in quasi-linear time. We believe[2] that the complexity is $O(|x| * D)$, where $|x|$ is the total size of both documents, and D the edit distance between them. The recent versions of DeltaXML support the addition of keys (either in the DTD or as attributes) that can be used to enforce correct matching (e.g. always match a *person* by its *name* attribute). DeltaXML also supports unordered XML trees.

Because Wu’s algorithm is applied at each level separately, the result is not strictly minimal. Note that real-world experiments showed that the result is in general (90 percent) strictly minimal. To understand the

[2]The algorithm has not been published

algorithm, we provide here an example where a non-minimal result is obtained.

```
First Document:
<top>
<a><b>text</b><b>text</b><b>text</b><b>text</b></a>
<a><b>text</b></a>
<a><b>text</b></a>
</top>

Second Document:
<top>
<a><b>new text</b></a>
<a><b>updated text</b><b>text</b><b>text</b><b>text</b></a>
<a><b>updated text</b></a>
<a><b>updated text</b></a>
</top>
```

The minimal edit script has 6 operations: 3 operations to update `text` nodes into `updated text`, and 3 to insert the `<a><b>text</b></a>` subtree (since it contains 3 nodes). Because the LCS algorithm is applied at each level, and does not consider the entire structure, `<a>` nodes are matched with no consideration of the text nodes content. For instance, the first `<a>` node on the second document is matched with the first `<a>` node is the first document, and so on. In our example, this leads to applying 17 operations on `<b>` nodes since the content below the pairs of matched `<a>` nodes differs significantly.

Logilab XmlDiff. In [51], Logilab proposes an Open Source XML diff tool. It represents changes using XUpdate or its own internal format. It offers two different algorithms to detect changes. For large files, it uses Fast Match Edit Script [52] (from S. Chawathe and al.). As previously, this algorithm applies a LCS computation (using Myer's algorithm) at each level and for each label. Thus, for the same reason as DeltaXML, it does not find the minimal edit script. It runs in $O(l * |D1| * e)$, where l is the number of node labels, and e the edit distance between the two documents.

The second algorithm is Zhang and Shasha's tree-to-tree correction algorithm (mentioned previously [43], [44]). It finds a minimal edit script considering insert and delete operations according to Tai's model (see Section II). More precisely, they improve Zhang Shasha algorithm by adding a *swap* operation between a node and its next sibling [11]. The complexity of this algorithm is quasi-quadratic.

Microsoft XmlDiff. Microsoft recently proposed an XML Diff and Patch toolset [53]. The source code is freely available. The delta format is Microsoft XDL. This tool is similar to **XML TreeDiff** [54] developed by

IBM and that was based on [55] and [43], [44]. Two diff algorithms are proposed. The first one, *Treewalk-XmlDiff*, is a fast tree-walking algorithm (in the spirit of Fast Match). To be fast, it uses a similar formula as XyDiff (see next) to limit the number of nodes visited during the tree walk. The second algorithm, *Precise-XmlDiff*, is an implementation of Zhang and Shasha algorithm [43], [44] (mentioned previously). Thus, the editing model is Tai's model (see Section II).

Before any of the two algorithms is used, a preprocessing phase is applied to the documents, which consists of matching identical subtrees (based on their hash signature) in the spirit of XyDiff (see next). Matched nodes are then removed, and the algorithm chosen is then applied on the pruned tree. This improves significantly the performance, and adds support for move operations (based on the matches during preprocessing). The drawback is that the delta is not always minimal.

```
Source Document   | Target Document   || Pruned Source | Pruned Target
                  |                   ||               |
<root>            |  <root>           || <root>        | <root>
  <a>             |    <a>            ||               |   <a>
    <x/>          |      <x2/>        ||               |     <x2/>
    <y/>          |      <x/>         ||               |     <x/>
  </a>            |      <y2/>        ||               |     <y2/>
  <a>             |      <y/>         ||  <a>          |     <y/>
    <x/>          |    </a>           ||    <x/>       |   </a>
    <x2/>         |    <a>            ||    <x2/>      |
    <y/>          |      <x/>         ||    <y/>       |
    <y2/>         |      <y/>         ||    <y2/>      |
  </a>            |    </a>           ||  </a>         |
</root>           |  </root>          || </root>       | </root>
```

Fig. 4. Two versions of a document

An example is shown in Figure 4 (page 25). The best (minimal) script is to move `x2` and `y2`, its cost is 2. However, when considering move operations, finding the minimum edit script is NP-hard. When move operations are ignored, some algorithms (e.g. Zhang-Shasha or MMDiff) find the minimum edit script. In this example, it consists of deleting `x2,y2` in the source document and then inserting `x2,y2` in the right place. Its cost is 4. Microsoft XmlDiff-Precise and IBM Treediff use such an algorithm, but their result is not minimal. The reason is that the pairs of identical subtrees in both documents are matched during the pruning phase. Thus, a different edit script is found. In this example, the two `<a><x/><y/></a>` subtrees are matched. On the pruned trees, i.e. without the matched subtrees, there is only one `<a>` subtree. The

ZhangShasha algorithm finds the minimal edit script on the pruned documents. It is to delete `x2,y2` and insert them before their previous sibling node `x,y`, within the same subtree. The cost is 4. Then, considering the first `<a>` matching, an additional move operation is necessary in order to swap the two subtrees, so the total cost is 5.

In next section, we consider algorithms that support move operations.

C. Tree pattern matching with a 'move' operation

The main reason why few *diff* algorithms supporting move operations have been developed is that most formulations of the tree diff problem are NP-hard. One may want to convert pairs of delete and insert operations into move operations. However, the result obtained is in general not minimal, unless the cost of move operations is strictly identical to the total cost of deleting and inserting the sutree.

LaDiff. Recent work from S. Chawathe includes *LaDiff* [52], [56], designed for hierarchically structured information. It introduces a matching criteria to compare nodes, and the overall matching between both versions of the document is decided on this base. A minimal edit script (according to the matching) is then constructed. Its cost is in $O(n * e + e^2)$ where n is the total number of leaf nodes, and e a weighted edit distance between the two trees. Intuitively, its cost is linear in the size of the documents, but quadratic in the number of changes between them. Note that in terms of worst-case bounds, when the change rate is large the cost becomes quadratic in the size of the data. Since we do not have an XML implementation of LaDiff, we could not include it in our experiments.

XyDiff. It has been proposed in [18]. This tool is free and Open Source, it is also used in the company Xyleme [2]. *XyDiff* is a fast algorithm which supports move operations and XML features like the DTD ID attributes. Intuitively, it matches large identical subtrees found in both documents, and then propagates matchings.

• *Propagating matchings.* Consider two `person` nodes that are already matched. We want to propagate matchings to descendant nodes. If each `person` node has a unique `firstname` child, it is easy to decide to match the two `firstname` nodes. In other cases, the situation may be more difficult. Consider for instance two `list-of-employees` nodes that are matched. Below each of them are hundreds of `employee` nodes. Between the two versions, some may have been deleted, inserted or moved. Thus, matching the `employee` nodes is difficult. To match them, one may want to use text data below them, such as the name of the employee. But then, it is difficult to decide which text data to use. For instance some text data may be less useful (e.g. the firstname) and using it may result in wrong matches. XyDiff has then to decide if matching should be propagated or not, there is a trade-off between missing possible matchings and generating false matchings.

• *Matching subtrees.* This is achieved in two ways. First, it uses the ID attributes that are possibly defined in the DTD since the value of an ID attribute uniquely identifies the node through time. Second, it matches subtrees that are strictly identical in both versions of the document. This is in the spirit of IBM and Microsoft's tools pruning phase. This is achieved by comparing hash signatures of the subtrees content. XyDiff starts by the largest subtrees and considers smaller and smaller subtrees if matching fails. The larger the matched subtrees are, the further the matching is propagated to ancestors of the subtree roots.

XyDiff has an upper bound cost (in time and space) that is proved [18] to be no more than $O(n * log(n))$, where n is the size of the documents (e.g. total size of files). However this algorithm does not, in general, find the minimum edit script.

D. Other Tools

We tested several other diff algorithms that we do not present here since they rely on similar algorithms as the previous ones. We also briefly mention some programs that we tested but did not work well for extensive testing on our large sets of XML data.

First is **XML treediff** [54] developed by IBM and which was based on [55] and [43], [44]. It was developped before, and is close to, the *Precise* version of Microsoft XmlDiff. It uses the same tree-edit algorithm (Zhang-Shasha) with a similar pruning phase.

Sun also released an XML specific tool named **DiffMK** [57] that computes the difference between two XML documents. This tool is based on the *GNU diff* algorithm (or Unix diff), and uses a *list* description of the XML nodes instead of a tree. In the same spirit, DecisionSoft proposes an Open Source xml diff program [35]. The program uses a linear representation of the XML document, i.e. the XML document is printed as text, and each printed line is considered as a node. Then, the Unix *diff* command is executed, and it finds which lines have been inserted or deleted. The performance (diff) and quality (deltas) of these two tools correspond exactly to the GNU diff tool.

The delta format they use is a text format.

E. Summary of tested diff programs

As previsouly mentioned, the algorithms are summarized in Figure 5 (page 29). The time cost given here (quadratic or linear) is a function of the data size. For *C3*, it corresponds to the case when there are few changes (i.e. $D << |x| + |y|$).

For GNU diff, we do not consider minimality since it does not support XML (or tree) editing operations. However, we mention in Section V-C some analysis of the result file size.

V. EXPERIMENTS

In this section, we present experiments that we conducted. First, we present measures on the sizes of deltas and conversions between delta formats. Then, we present performance measures (time and memory) on XML diff tools. Finally, we present some measures on the minimality of the results of various XML diff tools.

A. Representing Changes: delta formats

Conversion between formats. We implemented tools to convert the deltas from one format to another [58]. The main difficulty was to convert XDL diffgrams into other formats, since the edit model is different. As shown previously, the delta obtained are more complex than the initial diffgrams. The other conversions do not lead to difficulties. However, the conversions are often incomplete due to the support of different features by the formats. For instance, only XyDelta contains information about deleted nodes. Thus, converting to XyDelta results in missing data for delete operations. Conversely, converting from XyDelta to another format results in loss of information (the deleted content). Consequently, the formats are not strictly “equiva-

Fig. 5. Quick Summary

Program Name	**Author**	**Time**	**Memory**	**Moves**	**Minimal Edit Cost**	**Notes**
(C2) Algorithms finding the minimum edit script (no move)						
MMDiff	Chawathe and al.	quadratic	quadratic	no	yes	tests with our implementation
XMDiff	Chawathe and al.	quadratic	constant	no	yes	quadratic I/O cost tests with our implementation
Constrained Diff	K. Zhang	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping
X-Diff	Y. Wang, D. DeWitt, Jin-Yi Cai (U. Wisconsin)	quadratic	quadratic	no	yes	-for unordered trees -constrained mapping
XmlDiff ZhangShasha	Logilab	quadratic	quadratic	no	yes	
(C3) Fast algorithms based on LCS and other quadratic computations						
DeltaXML	DeltaXML.com	linear	linear	no	no	
GNU Diff	GNU Tools	linear	linear	no	-	no XML support (flat files)
XmlDiff Precise	Microsoft	quadratic	quadratic	yes	no	Zhang-Shasha (Pruning phase first)
XMLTreeDiff	IBM	quadratic	quadratic	yes	no	(see Microsoft XML Diff)
DiffMK	Sun	quadratic	quadratic	no	no	see GNU diff
XML Diff	DecisionSoft					see GNU diff
(C4) Linear time algorithms						
XmlDiff Treewalk	Microsoft	linear	linear	yes	no	
XyDiff	INRIA	linear	linear	yes	no	
XmlDiff FastMatch	Logilab	linear	linear	no	no	
others						
LaDiff	Chawathe and al.	linear	linear	yes	no	criteria based mapping
XML Diff	Dommitt.com					we were not allowed to discuss it

lent”. However, by modifying them slightly to support the same features (e.g. store the deleted content as in XyDelta), one may obtain equivalence.

Settings. We decided to compare the size of deltas. More precisely, we want to compare the size and effectiveness of the formats for the same sequence of operations. To do so, we compare deltas that have the same cost, i.e. that modify the same number of nodes. Consider a given set of operations transforming one document into another. We generate the corresponding delta in all formats. Our experiments were conducted over more than twenty thousand XML diffs, roughly twenty percent of the changing XML that we found on the web.

Size of deltas. First, we’ve measured that XyDeltas are smaller than XDL, which are themselves smaller than XUpdate. However, the relative ratio of size between these deltas is not important. The main difference between them is related to syntactical details. Such details are unimportant since the real delta size may vary depending on the storage model (file storage, native XML, relational XML storage).

The main difference between delta sizes is shown in Figure 6 (page 31). This Figure shows the space usage as function of the edit cost of the delta. The delta cost is defined according to the “1 per node” cost model presented in Section II. We distinguished between two different approches: list of operations (XyDelta, XDL, XUpdate) vs. summary of the document and its changes (DeltaXML). For clarity reasons, and without loss of generality, only XyDelta and DeltaXML sizes are displayed on the Figure. Each dot represents the average delta file size for deltas with a given edit cost. Note that although fewer dots appear in the left part of the graph, they represent each the average over several hundred measures. It confirms clearly that DeltaXML is larger for lower edit costs because it describes many unchanged elements. On the other hand, when the edit cost becomes larger, its size is comparable to XyDelta.

B. Detecting Changes: Speed and Memory usage

As previously mentioned, our XML test data has been downloaded from the web. The files found on the web are on average small (a few kilobytes). They contains various amounts of changes, and contain delete and insert operations, as well as changes in the order of some nodes. Real move operations are unfrequent. To run

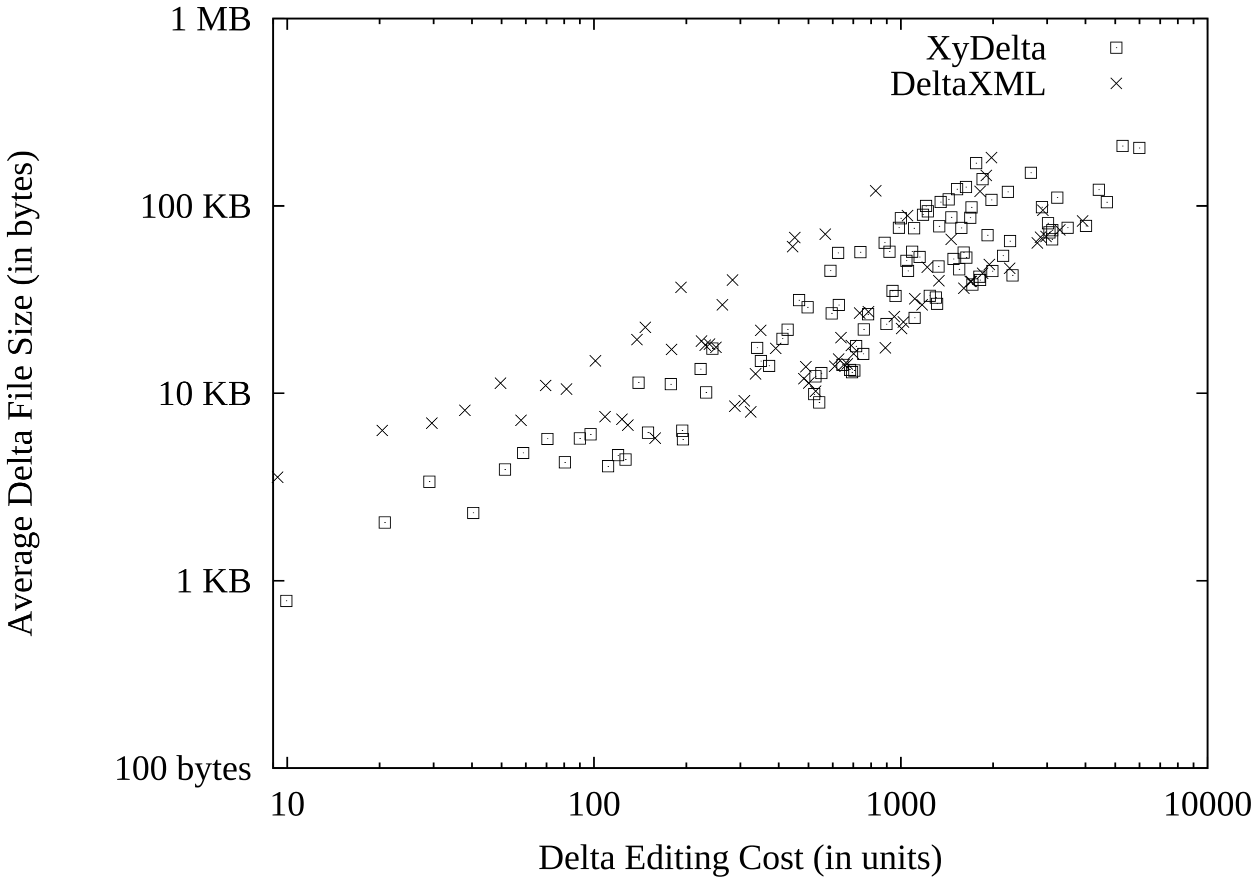

Fig. 6. Size of the delta files

tests on larger files, we composed large XML files by aggregating XML data from DBLP [17] data source. We used two versions of the DBLP source, downloaded at an interval of one year. The updates consists mainly in insertion of nodes since DBLP is a database of published papers. By mixing files and changing their order, we also obtained different types of changes. We also used our change simulator, which has been presented in [18]. It allowed us to modify parameters such as the frequence of insertion/deletion/update/move of nodes. Without loss of generality, the results presented in the next Figure are those obtained on the Web data set, composed of over 50000 changing XML documents, which we obtained by crawling 400 million web documents during one year.

Memory Usage. MMDiff, representing the quadratic cost algorithms, uses on average 1Gb of memory to compute the diff between two large files of a total 500kb size. XMDiff, which is an external-memory version of MMDiff, can run with limited memory but the I/O cost (for writting to disk) becomes quadratic. Thus, in our experiments, memory usage was clearly shown to be a limiting factor for quadratic algorithms.

On the other hand, fast and linear algorithms (C3 and C4) use no more memory than the size of the data. However, since they load the XML data in main memory, they are limited to documents of size below 100MB (depending on software versions). It would be interesting to have tools that run on a disk representation of the XML tree structure (e.g. Persistent-DOM).

Speed (Mean). The measures presented in Figure 7 were conducted on a Linux system. Some of the XML diff tools are implemented in C++, whereas others are implemented in Java. Let us stress that we ran tests that show that these algorithms compiled in Java (Just-In-Time compiler) or C++ run on average at the same speed, in particular for large files. The measures presented in Figure 7 confirm the classification of algorithms. Note that for clarity reasons we did not include all tested programs in this Figure, such as Logilab XmlDiff, Sun DiffMK or IBM treediff for instance.

Quadratic algorithms (e.g. MMDiff, XMDiff) run 1 hour to diff 1Mb of data, whereas 'fast' (e.g. DeltaXML) and 'linear' (e.g. XyDiff) algorithms run roughly 10 seconds. It should also be noted that our implementation of XMDiff, the external-memory version of MMDiff, is significantly slower than MMDiff due to I/O costs.

The *Treewalk* (i.e. linear) version of Microsoft XmlDiff is also ten times faster than XyDiff or DeltaXML for large files. This should be linked to experiments on XyDiff that showed that 90 percent of the time is spent in the XML parser. It seems that the .Net platform has here a significant advantage over Xerces C/C++ (XyDiff) or Java (JDK1.4) to read/parse XML data, however this is out of the range of our study.

GNU Diff is also much faster than others because it does not parse or handle XML. This makes *GNU Diff* very performant for simple text-based version management schemes. In some cases, the XML document may be printed as a single line. This is confusing for GNU diff that consider the lines of the documents. Sun DiffMK and DecisionSoft XmlDiff solve this problem by writing one line for each node, but they run much slower (speed is comparable to XyDiff).

The “quadratic” version of Logilab XmlDiff, IBM treediff and Microsoft XmlDiff *Precise* are not presented since we could not run them on entire set of test files. The reasons were some bugs and also their quadratic cost which is a bottleneck for medium/large files. Their running time was no less than MMDiff.

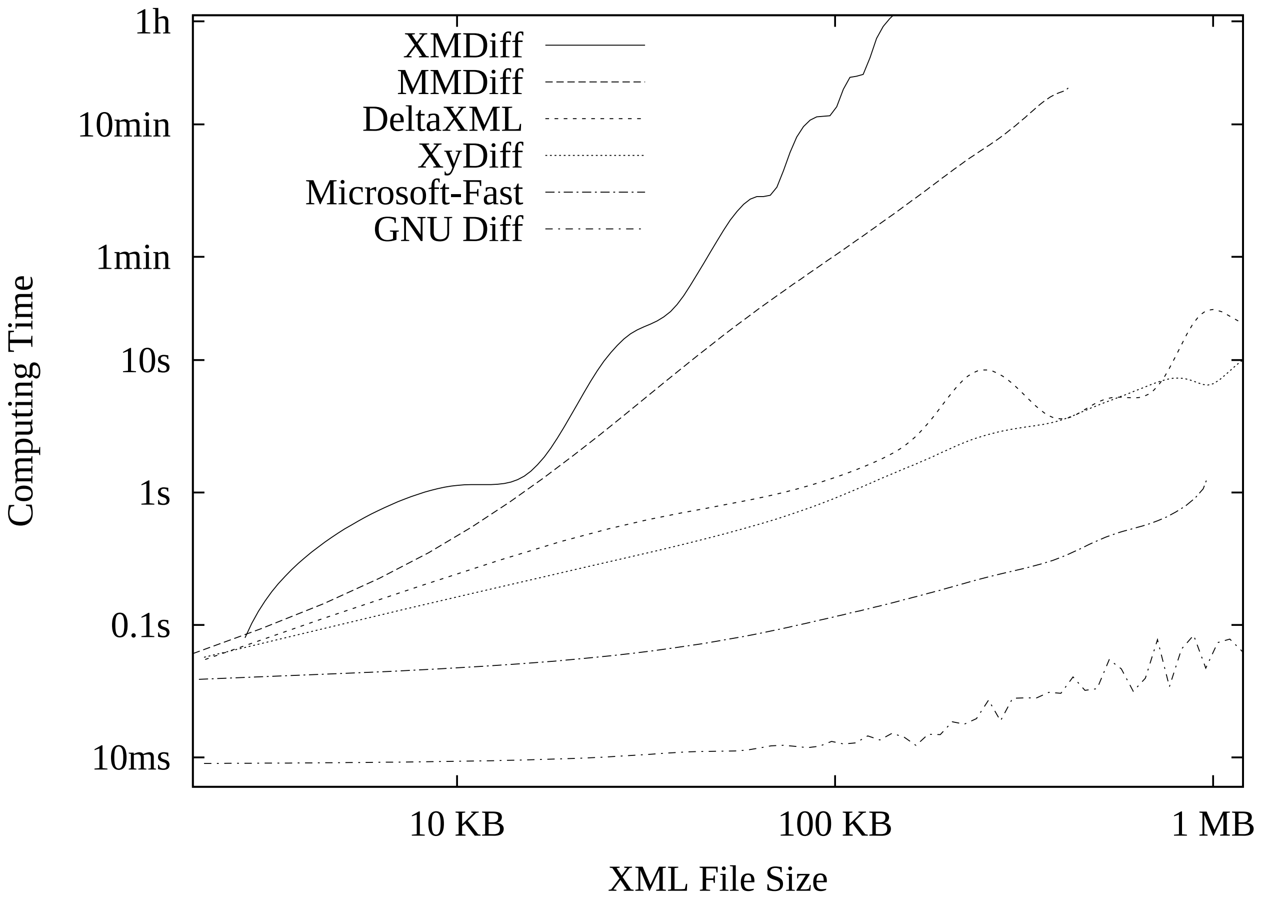

Fig. 7. Speed of different programs

The *Fast-Match* version of Logilab XmlDiff, somehow surprisingly, has roughly the same speed than MMDiff. It seems that their implementation of the LCS at each level may be improved for low change rates (e.g. D-Band).

Speed (Standard deviation). According to Figure 7, it seems that the speed of ’linear’ algorithms (e.g. XyDiff) is almost exactly the same as the speed of “fast” algorithms (e.g. DeltaXML). However, consider Figure 8 (page 34), a more detailed version that focuses on DeltaXML and XyDiff measures. It shows that the computation time for DeltaXML on some large documents is significantly longer. Indeed, it is roughly as long as the ‘quadratic’ computation time (recall MMDiff measures). This confirms that since “fast” algorithms are based (for some parts) on quadratic computations, there is a worst-case quadratic cost, although the average cost is almost linear. Experiments showed that this worst-case cost corresponds to higher change rate on the XML data. Indeed, the worst case for *D-Band* algorithms (see previous section) is when the edit distance *D* (i.e. the number of changes) between the two documents is close to the number of nodes N. In other words, when there are lots of changes the computation times becomes close to quadratic.

This may be a slight disadvantage for ’fast’ algorithm in applications with strict time requirements, e.g. computing the diff over a flow of crawled documents as in NiagaraCQ [22] or Xyleme [23]. On the contrary, ’linear’ algorithms such as *XyDiff*, Microsoft *XmlDiff-Fast* and *GNU Diff* have a very small standard deviation. This proves that their average complexity is close to the upper bound (i.e. linear).

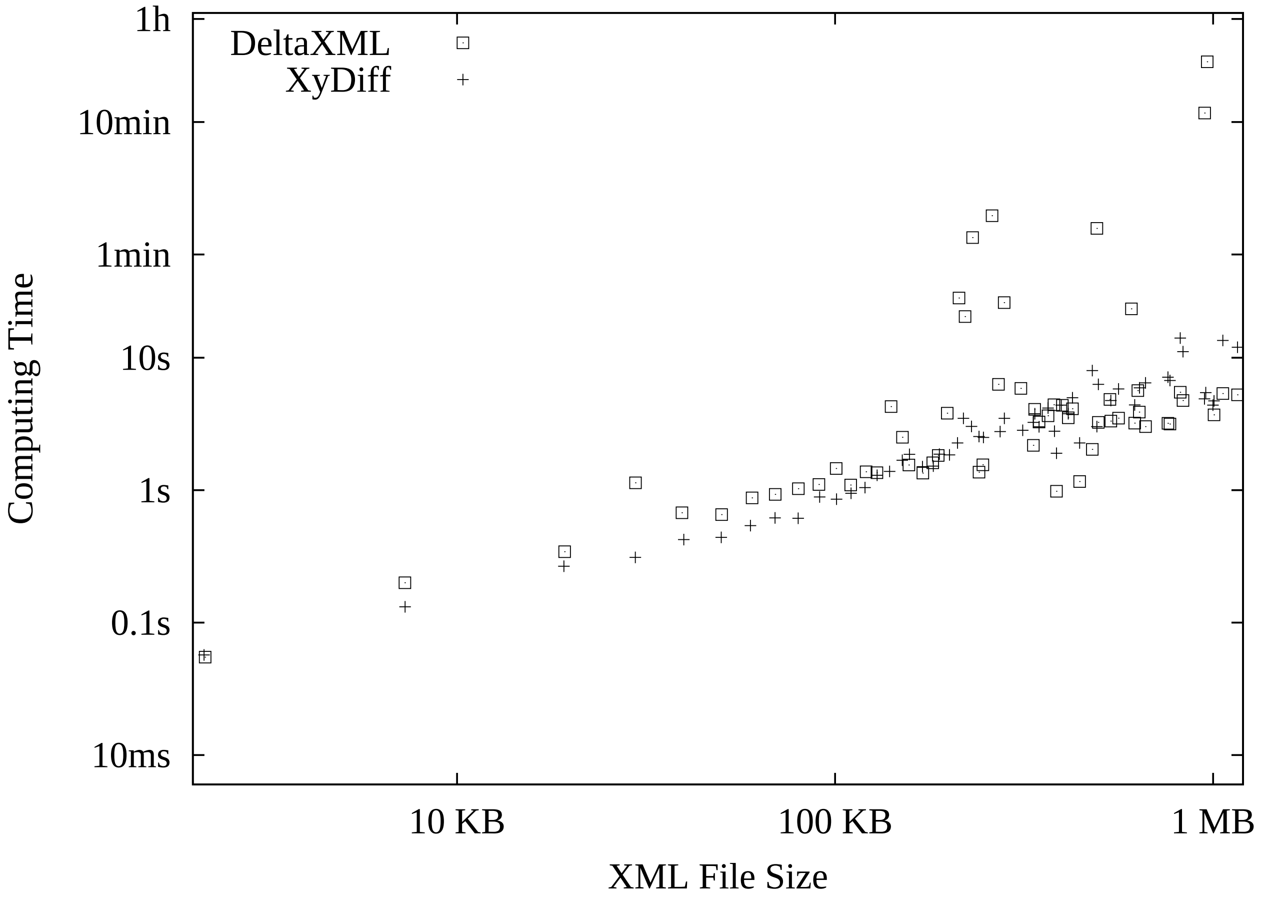

Fig. 8. Focus on DeltaXML speed measures

To summarize, there are orders of magnitude between the running time of linear vs. quadratic algorithms when handling medium files (e.g. hundred kilobytes). Linear and Fast algorithm have roughly the same average speed, but the worst-case for “fast” algorithms may be as costly as a quadratic computation. On the other hand, the difference between linear algorithms is mostly related to the cost of parsing XML data.

C. Detecting Changes: Minimality and quality of the result

The "quality" study in our benchmark consists of comparing the sequence of changes generated by the different algorithms. We recall that concise deltas are (in general) of better quality than others. We used the result of MMDiff (i.e. *XMDiff*) as a reference because these algorithms find the minimum edit script. Thus, for each pair of documents, the quality for a diff tool (e.g. DeltaXML) is defined by the ratio

$$r = \frac{C}{C_{ref}}$$

where C is the delta edit cost and C_{ref} is MMDiff delta's edit cost for the same pair of documents. A quality equals to one means that the result is minimum and is considered "reference". When the ratio increases, the quality decreases. For instance, a ratio of 2 means that the delta is twice more costly than the minimum delta. Ratio from 1 to 2 are named *Good* in our graph. Ratios from 2 to 5 are named *Medium* in our graph. There was almost no delta with ratio over 5. In Figure 9 (page 36), we present an histogram of the results, i.e. the (percent) number of documents in some range of quality. The experiment was run over 50000 different deltas, as explained previously. *XMDiff* and MMDiff are the reference, thus all their deltas have a quality strictly equal to one. *GNU Diff* do not appear on the graph because it does not construct XML (tree) edit sequences. We also conducted experiments with *Microsoft Diff* and *Logilab Diff*. The *Precise* version of Microsoft XmlDiff and the quadratic version of Logilab did performs roughly the same as DeltaXML, finding slightly smaller deltas due to their edit model. However, they could not support larger files. The *Treewalk* version of Microsoft XmlDiff performed roughly the same as XyDiff.

These results in Figure 9 show that:

- (i) DeltaXML: For most of the documents, the quality of *DeltaXML* results is as good as the reference (strictly equal to 1). For the others, the delta is on average thirty percent more costly than the minimum, which is very good.
- (ii) XyDiff: Almost half of the deltas are less than twice more costly than the minimum. The other half costs on average three times the minimum. Since XyDiff supports move operations, which may replace a pair of insert/delete at a lower cost, some of the delta (roughly 18 percent) have a better cost than the reference

(see category *Better* in the Figure).

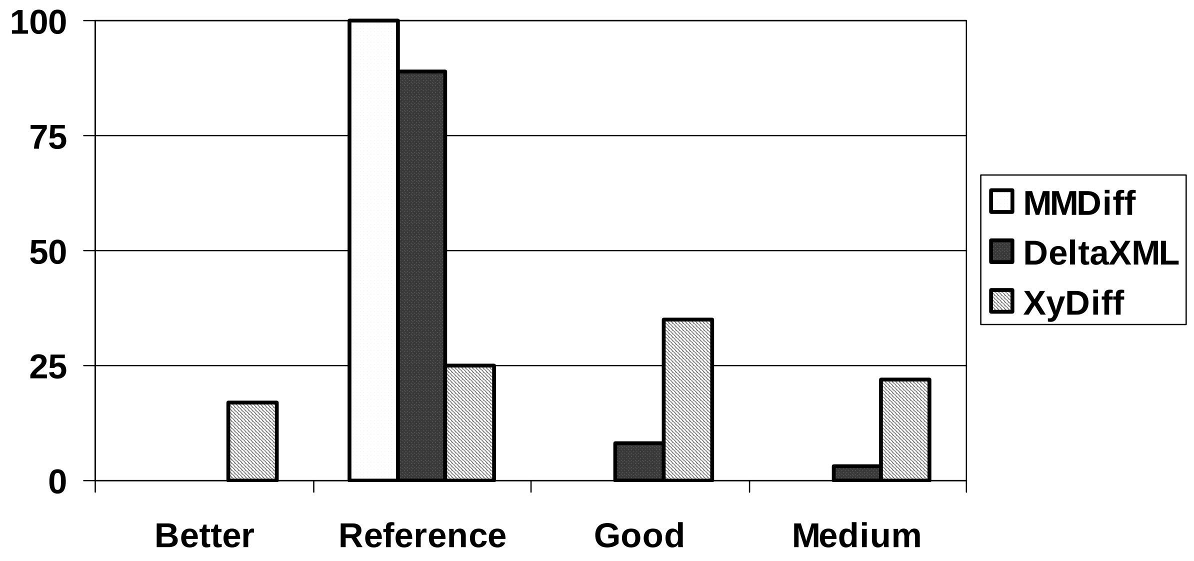

Fig. 9. Quality Histogram

Delta files size. In order to compare the deltas that use different edit models (e.g. Tai's model for Microsoft XmlDiff, text-model for GNUDiff), we also compared the size of the delta files. However, we do not present detailed results since the comparison of file sizes may be biased by other factors such as the delta formats used and the amount of additional information in the delta.

Again, we considered the reference to be the deltas from MMDiff, using the XyDelta format for representing changes. Although DeltaXML deltas have almost the same cost as MMDiff deltas, there are on average fifty percent larger due to the storage overhead of the DeltaXML delta format, in particular for low change rates. For the same reason, XyDiff deltas (using XyDelta format) have a higher edit cost but have on average the same size as DeltaXML deltas. The size of Microsoft XmlDiff *Treewalk* deltas (using Microsoft XDL format) is also similar, since their cost (using Tai's model) is similar to XyDiff. The *Precise* versions found even smaller deltas, but could not support large files. An important aspect is to note that GNU diff deltas (i.e. text deltas) are also roughly the same size (perhaps closer to MMdiff).

Finally, note that the quality is measured here by the support for XML tree operations and the conciseness of deltas. However, some tools may provide richer semantics such as *matching rules*. For instance, XyDiff supports ID attributes defined in the DTD, and DeltaXML allows to define keys on the XML data. More work is clearly needed in the direction of evaluating the semantic quality of results. We also intend to conduct experiments on *LaDiff* [52] which is a good example of criteria-based mapping and change detection.

VI. Related Work

Previous work is abundant on the topic of diff and tree-diff algorithms. A detailed state of the art on tree-diff algorithms is presented in [11]. This work also presents the tree-to-tree correction problem, edit formalisms as well as tree edit distance computation. However, it does not consider more recent algorithms such as [46], [52], and in particular it does not consider the algorithms that run on XML data. Also it does not present experiments. [20] consider version management in XML data with keys. Their work is focused on comparing the efficiency of storage models for large sequences of changes. They compare various storage strategies for sequences of deltas (e.g. last version and all deltas, compressing deltas, aggregating deltas), and propose a model that annotates the nodes in the original document with time validity information. The main differences with our work are: (i) we consider the problem of change detection, whereas their approach relies on simple change detection schemes that use “keys” defined on the XML document, (ii) we compare delta formats, and their efficiency for monitoring/querying changes, whereas they do not study the delta formats themselves but rather the strategies for storing long sequences of deltas. In order to query multiversion XML document, [21] addresses the persistent identification of nodes using numbers that encode structure information.[19] studies storage strategies for managing large numbers versions, it also studies physical level schemes. Up to our knowledge, there is no previous work that compares XML diff tools and/or XML delta formats.

VII. Conclusion

In this paper, we described existing works on the topic of change detection in XML documents.

We first presented recent proposals for representing changes, and compared their features through analysis and experiments. We believe that more work is needed to propose a framework for version management and querying changes according to the proposed change languages. There is a trade-off between having lots of information in the deltas (e.g. DeltaXML, XUpdate) and having more compact deltas (e.g. XyDelta, XDL). Since a standard for querying changes is still missing, having deltas with lots of information is for the moment a simple way to query/monitor changes. However, the long-term focus should be more on using compact deltas and using persistent identifiers to process temporal queries. Thus, persistent identifiers (as in XyDelta) are an important aspect, delta formats should be extended to support it. Some important features should also be considered, such as the support of move operations (e.g. XDL, XyDelta) or backward deltas. At the cost of little improvements, the languages presented here are almost equivalent. More work is clearly needed to define a common standard or refine the formal model of existing proposals.

The second part of our study concerns change detection algorithms. First, we underlined the importance of the quality and semantics of results. The study and experiments presented show (i) a significant quality advantage for “quadratic” and “fast” algorithms (C2 and C3, see MMDiff, DeltaXML and Microsoft Precise-XmlDiff) (ii) a dramatic performance improvement with linear complexity algorithms (C4 and some C3, see GNU Diff, Microsoft TreeWalking-XmlDiff, XyDiff, DeltaXML).

On one hand, only MMDiff (and XDiff for unordered trees) finds the exact minimal edit script, but it does not scale to large files (e.g. 1Mb). On the other hand, some algorithms (Microsoft Treewalk, XyDiff) always run in linear time, but the quality of their results is lower. DeltaXML runs on average in linear time, but the cost may be quadratic for some files. It often finds the minimal delta. Thus, DeltaXML seems a good compromise and it is currently one of the most reliable tools. We also noted that flat text based diff (e.g. GNU Diff) may be used to efficiently store and version XML data. No algorithms have been proposed that run on streaming XML data. Also we did not find tools that handle files above 100Mb or 1Gb. Such tools may be implemented by processing an on-disk representation of the XML data, more work is needed in that direction. An important aspect is also to improve the semantics of results. For instance, DeltaXML and XyDiff allow to define *keys* on the XML data to identify nodes. This may be useful in many applications where results must have precise semantics. Also it improves performance since nodes with keys are matched very quickly. More work is needed in that direction.

Although the problem of “diffing” XML (and its complexity) is better and better understood, there is still room for improvement. In particular, diff algorithms could also take advantage of semantic knowledge that we may have on the documents or may have infered from their histories or their DTD.

Acknowledgments We would like to thank Serge Abiteboul, Vincent Aguiléra, Robin La Fontaine, Amélie Marian, Tova Milo, Benjamin Nguyen and Bernd Amann for discussions on the topic.

References

[1] W3C, “EXtensible Markup Language (xml) 1.0,” http://www.w3.org/TR/REC-xml.
[2] “Xyleme,” `www.xyleme.com`.
[3] W3C, “XQuery,” `www.w3.org/TR/xquery` .
[4] “XPath, XML Path Language,” `http://www.w3.org/TR/xpath/` .
[5] V. Aguiléra, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez, “Querying XML Documents in Xyleme,” in *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, july 2000.
[6] Sophie Cluet, Pierangelo Veltri, and Dan Vodislav, “Views in a large scale XML repository,” in *The VLDB Journal*, 2001, pp. 271–280.
[7] W3C, “Resource description framework,” www.w3.org/RDF.
[8] FSF, “GNU diff,” www.gnu.org/software/diffutils/diffutils.html.
[9] “XML Schema,” `http://www.w3.org/XML/Schema/` .
[10] “Document Object Model (DOM),” `http://www.w3.org/DOM/` .
[11] David T. Barnard, Gwen Clarke, and Nicolas Duncan, “Tree-to-tree Correction for Document Trees David T. Barnard,” in *Technical Report 95-372, Department of Computing and Information Science, Queen’s University, Kingston*, 1995.
[12] S. Chawathe, S. Abiteboul, and J. Widom, “Representing and querying changes in semistructured data,” in *ICDE*, 1998.
[13] R. La Fontaine, “A Delta Format for XML: Identifying changes in XML and representing the changes in XML,” in *XML Europe*, 2001.
[14] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet, “Change-centric Management of Versions in an XML Warehouse,” *VLDB*, 2001.
[15] XML DB, “Xupdate,” http://www.xmldb.org/xupdate/.
[16] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo, “Version management of XML documents,” *Lecture Notes in Computer Science*, 2001.
[17] Michael Ley, “Dblp,” dblp.uni-trier.de/.
[18] G. Cobena, S. Abiteboul, and A. Marian, “Detecting changes in XML documents,” in *ICDE*, 2002.
[19] S-Y. Chien, V.J. Tsotras, and C. Zaniolo, “Efficient Schemes for Managing Multiversion XML Documents,” *VLDB Journal*, vol. 11, no. 4, pp. 332–353, 2002.
[20] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan, “Archiving scientific data,” in *SIGMOD Conference*, 2002.
[21] S.-Y. Chien, V.J. Tsotras, C. Zaniolo, and D. Zhang, “Efficient Complex Query Support for Multiversion XML Documents,” in *EDBT*, 2002.
[22] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang, “NiagaraCQ: a scalable continuous query system for Internet databases,” in *SIGMOD*, 2000.
[23] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda, “Monitoring XML data on the web,” in *SIGMOD*, 2001.
[24] K.C. Tai, “The tree-to-tree correction problem,” in *Journal of the ACM, 26(3)*, july 1979, pp. 422–433.
[25] S. M. Selkow, “The Tree-to-Tree Editing Problem,” *Information Processing Letters, 6*, pp. 184–186, 1977.
[26] Jason T.L. Wang, Bruce A. Shapiro, Dennis Shasha, Kaizhong Zhang, and Kathleen M. Currey, “An algorithm for finding the largest approximately common substructures of two trees,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 8, pp. 889–895, 1998.
[27] Kaizhong Zhang, J. T. L. Wang, and Dennis Shasha, “On the editing distance between undirected acyclic graphs and related problems,” in *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, 1995, pp. 395–407.
[28] S. Abiteboul, P. Buneman, and D. Suciu, “Data on the web: From relations to semistructured data and XML,” *Morgan Kaufmann Publisher*, October 1999.
[29] Microsoft, “XDL: XML Diff Language,” http://www.gotdotnet.com/team/xmltools/xmldiff/.
[30] E. Cohen, H. Kaplan, and T. Milo, “Labeling dynamic XML trees,” in *PODS*, 2002.
[31] Z. Vagena and V.J. Tsotras, “Path-expression Queries over Multiversion XML Documents,” in *WebDB*, 2003.
[32] G. Cobena, “Change Management for semi-structured data on the Web,” *PhD thesis*, 2003, `http://www-rocq.inria.fr/gemo/` .
[33] S. S. Chawathe, S. Abiteboul, and J. Widom, “Managing Historical Semistructured Data,” *Theory and Practice of Object Systems*, vol. 5, no. 3, pp. 143–162, Aug. 1999.
[34] Dommitt Inc., “XML diff and merge tool,” www.dommitt.com.
[35] Decision Soft, “XML diff,” http://tools.decisionsoft.com/xmldiff.html.
[36] A. Apostolico and Z. Galil, Eds., *Pattern Matching Algorithms*, Oxford University Press, 1997.
[37] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Cybernetics and Control Theory 10*, pp. 707–710, 1966.
[38] D. Sankoff and J.B. Kruskal, “Time warps, string edits, and macromolecules,” *Addison-Wesley, Reading, Mass.*, 1983.
[39] R.A. Wagner and M.J. Fischer, “The string-to-string correction problem,” *Jour. ACM 21*, pp. 168–173, 1974.
[40] W.J. Masek and M.S. Paterson, “A faster algorithm for computing string edit distances,” in *J. Comput. System Sci.*, 1980.
[41] Eugene W. Myers, “An O(ND) difference algorithm and its variations,” in *Algorithmica*, 1986.
[42] S. Wu, U. Manber, and G. Myers, “An O(NP) sequence comparison algorithm,” in *Information Processing Letters*, 1990, pp. 317–323.

[43] Kaizhong Zhang and Dennis Shasha, "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," *SIAM Journal of Computing, 18(6)*, pp. 1245–1262, 1989.

[44] Kaizhong Zhang and Dennis Shasha, "Fast algorithms for the unit cost editing distance between trees," *Journal of Algorithms, 11(6)*, pp. 581–621, 1990.

[45] W. Yang, "Identifying syntactic differences between two programs," *Software - Practice and Experience, 21, (7)*, pp. 739–755, 1991.

[46] S. Chawathe, "Comparing Hierarchical Data in External Memory," in *VLDB*, 1999.

[47] Kaizhong Zhang, R. Statman, and Dennis Shasha, "On the editing distance between unordered labeled trees," *Information Proceedings Letters 42*, pp. 133–139, 1992.

[48] Kaizhong Zhang, "A constrained edit distance between unordered labeled trees," in *Algorithmica*, 1996.

[49] Yuan Wang, David J. DeWitt, and Jin-Yi Cai, "X-diff: A fast change detection algorithm for xmldocuments," in *ICDE*, 2003.

[50] DeltaXML, "Change control for XML in XML," www.deltaxml.com.

[51] Logilab, "XML diff," http://www.logilab.org/xmldiff/.

[52] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD*, vol. 25, no. 2, pp. 493–504, 1996.

[53] Microsoft, "XML Diff and Patch," http://www.gotdotnet.com/team/xmltools/xmldiff/.

[54] IBM, "XML treediff," www.alphaworks.ibm.com/.

[55] F.P. Curbera and D.A. Epstein, "Fast difference and update of xml documents," in *XTech*, 1999.

[56] S. Chawathe and H. Garcia-Molina, "Meaningful Change Detection in Structured Data," in *SIGMOD*, Tuscon, Arizona, May 1997, pp. 26–37.

[57] Sun Microsystems, "Making All the Difference," http://www.sun.com/xml/developers/diffmk/.

[58] "Diff XML,"
`http://www-inf.enst.fr/˜talel/rech.html` .