Merging Changes in XML Documents Using Reliable Context Fingerprints

Sebastian Rönnau Christian Pauli Uwe M. Borghoff

Institute for Software Technology
Universität der Bundeswehr München
Werner-Heisenberg-Weg 39
85577 Neubiberg, Germany
Sebastian.Roennau@unibw.de

ABSTRACT

Different dialects of XML have emerged as ubiquitous document exchange formats. For effective collaboration based on such documents, the capability to propagate edit operations performed on a document is indispensable. In order to avoid the transmission of whole documents, deltas are used to describe these edit operations, allowing the construction of a new version of a document. However, patching a document with a delta it was not generated for is error-prone, and any insert or delete operations performed on the document are likely to affect all subsequent paths within that document.

In this paper, we present a delta format for XML documents that uses context-aware fingerprints to identify edit operations. This allows our XML patch procedure to find the correct position of an edit operation, even if the document was updated in the meantime. Possible conflicts are detected. Experimental results show the reliability of the presented fingerprinting technique and prove the high quality of the resulting patched documents.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document and Text Editing—*Document management, Version control*

General Terms

Algorithms, Management, Reliability.

Keywords

CSCW, XML diff, XML patch, fingerprint, office applications, version control.

1. INTRODUCTION

In office work, collaborative editing of documents is an every-day task. Several persons write seperate parts, other persons review and comment them. Many different tools and systems try to facilitate the exchange of documents, taking technical and organizational measures to guarantee a consistent state. This belongs to the research body known as computer supported cooperative work (CSCW).

For office applications, XML has emerged as lingua franca. OpenOffice, Microsoft Office, and many other applications use XML dialects for serializing and exchanging documents. Therefore, in this paper we consider XML documents only.

Several metrics exist for categorizing XML-aware CSCW systems. Among others, they can be divided in operation-based and state-based systems [12, 13]. The main advantage of the operation-based approach is that it retains information about the evolution of a document state, thus allowing to perform a fine-grained merge. However, in the office domain this advantage turns into a major drawback: Persons (or organizations) often want to hide their editing process, only accepting to exchange a document in an approved state. Therefore, we focus our work on the state-based approach.

Version control systems can serve as an example for a state-based CSCW system. The question, how version control systems and XML-based office documents interact has been discussed in [25], where an XML-aware diff tool is used to compute the changes between two versions of a document. Several implementations of such a tool have been proposed. However, only two approaches are sufficiently efficent [7, 20].

Two-way-diffs can only be applied if documents are evolved in a linear fashion. In most collaboration and versioning scenarios however, it is crucial to be able to merge changes performed independently on a document [1]. As a solution, a three-way-diff could be used, which compares the changed versions of a document with their nearest common ancestor [13]. An XML-aware implementation was proposed, too [18]. However, these approaches require all three versions to be available. In ad-hoc environments and loose collaboration sytems, this precondition will not hold in general [24]. Low-bandwidth connections, for example over satellite systems, do not allow the transfer of the complete version, either.

Another solution would be to apply a delta to a version of the document it was not computed for – an approach which is commonly used in the domain of line-based edit operations [11]. Due to the fact that deltas mostly use absolute paths to identify an edit operation, this approach is both naive and unusable in the domain of XML documents. Figure 1 shows a simple example, where an edit operation affects the addresses of the subsequent nodes.

In order to avoid such effects, we present a technique to compute fingerprints of the context of an edit operation us-

DocEng'08, September 16–19, 2008, São Paulo, Brazil.

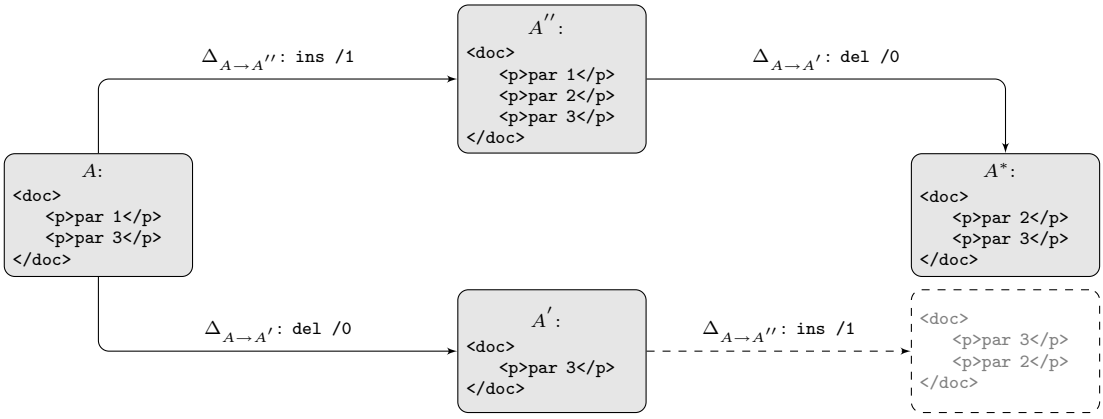

Figure 1: Applying a delta to a document version it was not computed for leads to unwanted results easily. Performing the delta marked with a dashed line would create a wrong document version.

ing hash values. This allows the patch procedure to identify the correct position of an edit operation using its context – even if it has moved in the meantime.

Thus, patching an XML document with a delta it was not computed for becomes possible and reliable.

The remainder of this paper is organized as follows: We define our XML model and a delta model in Section 2. In Section 3, we propose a fingerprinting technique using the context of an edit operation and a delta format using it. A patch procedure based upon this delta format is described in Section 4. Section 5 demonstrates the benefits of our approach using experimental results. After an examination of related work in Section 6, we conclude the paper and give an outlook on future work in Section 7.

2. PRELIMINARIES

The basic set of all documents shall be denoted as $\mathbb{A}$. A is a document of $\mathbb{A}$. A', A'', and A^* are versions of A.

2.1 XML Model

It must be emphasized that with our approach the order of nodes in the XML tree is significant. We use the term *document order* for the order in which nodes are encountered, one after another, as the document that contains them is parsed [6]. Hence, for two nodes $i, j \in A$, $i < j$ is true, if and only if i comes before j in document order.

In this paper, we use the term *node* for text, element and processing instruction nodes. Attributes are regarded as part of element nodes; comments are ignored, as they do not affect the semantic meaning of the document. The terms *ancestor*, *child*, *descendant*, *parent*, and *sibling* are used as defined in [6], too. A *subtree* is defined as exactly one node (called *root(subtree)*), which can have any descendant nodes without restriction. It is important to distinguish this document-centric view from a data-centric view, which is common in the domain of databases (see, e.g. [17]).

Empty text nodes and text nodes only consisting of whitespace are not taken into account by our approach. This decision was motivated by the intention to increase the information value of the context of an edit operation, as well as to be robust against pretty-printing of output (see Section 3.2).

We define the distance of two nodes $i, j \in A$ to be the number of nodes to walk in document order from i to j:

$$\mathrm{dist}(i,j) = \begin{cases} \#\{n \in A | i < n \leq j\} & \text{if } i < j \\ -\#\{n \in A | j < n \leq i\} & \text{if } j < i \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

A key question in the design of an XML delta format is how to define the address of an edit operation. With XPath, a powerful language for addressing nodes within XML documents exists [6]. However, an XPath expression typically returns a set of nodes, whereas an edit must act on a unique node. This problem can be avoided, since XPath allows to address single nodes on a hierarchy level using an absolute index like `[position()=1]`. Note that this absolute index must be defined on each hierarchy level to ensure that only one node is addressed. For example, `/doc/text/itemization/item[position()=2]` would return more than one item if the text contains multiple itemizations.

In our approach, we use a slightly simpler addressing syntax [20], where the expression `/0/1` equals the XPath expression `/*[position()=1]/*position()=2]`[1].

2.2 Delta Model

Two versions of one document, denoted as $A, A' \in \mathbb{A}$, are compared by a diff-algorithm, which computes the differences and creates an edit script:

$$\mathrm{diff} : \mathbb{A} \times \mathbb{A} \to \Delta \quad (2)$$

where Δ is the basic set of all edit scripts. An element of Δ, denoted by $\Delta_{A\to A'}$, contains the script which must be performed to construct the new version (A') from the given one (A) using a so-called patch-program:

$$\mathrm{patch} : \mathbb{A} \times \Delta \to \mathbb{A} \quad (3)$$

In the following, the term *delta* is used synonymously for an edit script, meaning an element of Δ.

Thus, a delta is a set of edit operations which must be performed to construct a new version of a document. Formally,

[1]As you can see, XPath begins its numbering with 1. In contrast, the proposed addressing scheme starts numbering with 0, which is more familiar to computer scientists anyway.

an edit operation is denoted by a tuple

$$\text{operation} : (type, position, v, v') \quad (4)$$

Its parameter items are described below.

2.2.1 Type of Edit Operations

The first item refers to the *type* of the edit operation, where the main types are "insert", "delete", and "update".

Basically, every change between two documents can be represented by a sequence of insert and delete operations. Still, we consider the update operation to be important for the following reason: Imagine that the root node of a text document contains an attribute declaring the document status as "draft" or "final". In order to effect a simple change of status without an update operation being available, one would have to first delete the whole document and to insert it again with the new status. This would result in a delta twice the size of the original document and all benefits of an XML-aware diff and patch would be lost.

Some approaches additionally make use of the operation types "move" [20, 7, 5], and "glue" [5]. These operation types would demand an extended notation and more complex patch rules which are both beyond the scope of this paper.

2.2.2 Addressing Changes

The *position* addresses a node in the document A where the edit operation will be performed. Later on in this paper, the term *designated point of operation* is used as a synonym for the *position* defined in an edit operation.

To state this more precisely, *position* addresses a node depending on the type of the edit operation.

- For insert operations, the *position* refers to the address, where the subtree to be inserted will be placed. An already existent node at this address and all of its following siblings will be shifted one position to the right. A subtree designated to be inserted as the last element of a subtree would also refer to the address where it will be placed, even if the address did not exist in the original document A (see Figure 2).

- In case of delete operations, the *position* addresses the root node of the subtree to be deleted in document A.

- An update operation addresses the node to be updated in document A.

Note that insert and delete operations address a subtree, whereas an update operation just addresses a single node. An example clarifies this: A delete operation deletes the addressed node itself and all of its children. If a delete operation would only delete the node addressed, all its children would have to move up one hierarchy level higher. We assume that this is probably not in conformance with a DTD, Schema or similar. Vice versa, an insert operation is not allowed to insert a node as a parent of existing nodes in document A. This behavior conforms with [5, 21], but is contradictory to [18, 20].

2.2.3 Value

The content of the area affected by the edit operation before and after its application is stored in v and v'. Both

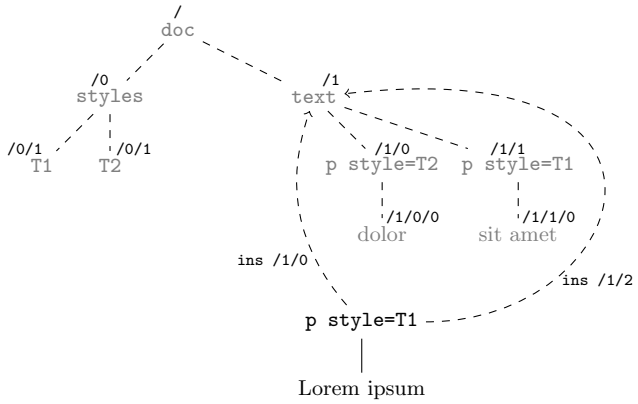

Figure 2: An insert operation addressing `/1/0` would shift the existing node at this position to the right, including all subsequent nodes w.r.t. the parent node `text`, and force the re-calculation of their paths. To insert the subtree as last child of `/1`, it must address `/1/2`.

refer to a node or a subtree, depending on the type of the edit operation[2].

- In case of an insert operation, v remains empty, and v' describes the subtree to insert.

- For delete operations, v contains the subtree to delete, whereas v' is empty.

- An update operation addresses single nodes. The old value of the node is stored in v, the new one in v'

This definition ensures the completeness of the delta in terms of [21], allowing a delta to be used both ways to construct a new version, and to revert the changes made (see Section 3.4).

2.2.4 Commutativity of Edit Operations

Finally, all edit operations within one delta have to be commutative. This results from the set property of the delta, which does not specify the order of the edit operations, thus allowing any permutations of operations.

Definition 1. *Two edit operations $op_1, op_2 \in \Delta$, with $op_1 \neq op_2$ are commutative, if the position of op_1 does not depend on the position of op_2 itself or a descendant of op_2 and vice versa.*

Commutativity ensures that an edit operation can possibly be performed, even in a situation where another one has been rejected due to a conflict (see Section 4.2). Note that this precondition can only be stated in a state-based model. Operation-based approaches cannot ensure the commutativity of operations.

Each edit operation within the same delta will be performed only once, and only on one node. An edit operation must not address multiple nodes (like template rules in XSLT can do).

[2]In terms of GNU diff, v and v' would be called a *hunk*.

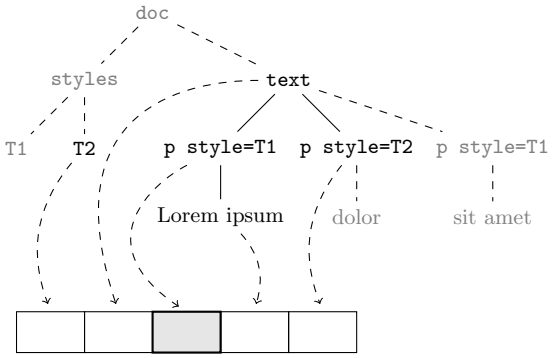

Figure 3: This fingerprint with radius $r = 2$ stores context-information for an update operation. The *anchor* node is the first paragraph, located at `/1/0`. For insert and delete operations addressing this node, the resulting fingerprint would differ slightly.

3. CONTEXT-ORIENTED DELTA

In general, documents have an order on a higher, non-technical level of abstraction. For example, the sequence of paragraphs within a text document is significant, even though no formal order on them would be defined.

We assume that most edit operations performed on a document are context-sensitive. A person working on a text for example, would insert a paragraph semantically relating to the paragraph before. An insertion at the right absolute position but in context of another paragraph would be highly arguable. Therefore, we assume that the context of an edit operation is a reliable indication, whether an edit operation should be performed or rejected.

To describe the context of an edit operation we introduce a fingerprinting technique in Section 3.1, which uses a hashing scheme presented in Section 3.2. The resulting delta format is described in Section 3.3. Finally, we show the completeness of the delta by presenting inversion rules in Section 3.4.

3.1 Fingerprinting Edit Operations

The context of an edit operation is stored in a so-called *fingerprint*, which is a sequence of the hash values of all nodes in a radius r ($r \geq 0$) around the edit operation in document order.

To be able to define a meaningful fingerprint, we introduce the context of an edit operation, called $A_{\mathit{fingerprint}}$:

$$A_{\mathit{fingerprint}} \stackrel{\text{def}}{=} A \backslash \mathit{descendants}(v) \tag{5}$$

As the fingerprint is defined using the document order, this definition ensures that the fingerprint does not refer to itself in a delete operation. The fingerprint around the *designated point of operation* i is a sequence, ordered by the distance relating to i:

$$\mathit{fingerprint}(i) = \{\text{hash}(j)\}, \text{where } j \in A_{\mathit{fingerprint}} \wedge 0 < |\mathit{dist}(i,j)| \leq r \tag{6}$$

The element of the fingerprint with distance d relating to i is denoted as $\mathit{fingerprint}_i[d]$. The value of $\mathit{fingerprint}_i[0]$ is denoted as *anchor* node and is assigned a value depending of the type of the edit operation:

$$\mathit{fingerprint}_i[0] \stackrel{\text{def}}{=} \begin{cases} \text{hash}(\mathit{root}(v')) & \text{for insert operations} \\ \text{hash}(\mathit{root}(v)) & \text{otherwise} \end{cases} \tag{7}$$

This ensures that the anchor of a fingerprint always refers to the first node of the area affected by an edit operation, even for insert operations, as they do not have an explicit node in A, which they could address. By this, edit operations are allowed to be inverted, as described in Section 3.4. For delete and update operations, the anchor node is used for a fast decision, whether a conflict occurs (see Section 4.2).

Figure 3 shows an example of a fingerprint which reveals the consequences of introducing $A_{\mathit{fingerprint}}$. An update operation referring to node `/1/0` would result in the fingerprint as shown. For a delete operation, the fingerprint would not include the descendants of the node to delete, which affects the right part of the shown fingerprint. Instead, `p style=T2` and `dolor` would be used for computing the fingerprint right of the anchor. An insert operation on the node `/1/0` would mean to "insert a subtree as the first child of `text`, pushing `p style=T1` and its siblings to the right". Therefore, `p style=T1` and `Lorem ipsum` would be used as right context. The anchor node would refer to the node to insert, which is not shown in this example.

If there do not exits enough nodes to fill the fingerprint, which happens near the document borders, a special "null node" is used, which is identified by its unique hash value.

The fingerprint has a size of $n = 2r + 1$, with the anchor node in the middle of the sequence. The computation of the hash values will be discussed in Section 3.2. According to the definition of *fingerprint* as a sequence, hash values are allowed to occur multiple times.

Note that the fingerprint basically does not contain any information about the address or position of the edit operation within the document tree. As a result, it would be possible to change the hierarchy level in our example by introducing a new parent node for `styles` and `text`, without affecting the fingerprint.

3.2 Reliable XML Hashing

A key property of reliable hash functions is the recognition of slight changes. In the context of XML, hashing is not trivial, as some properties of XML have to be considered.

Many ways exist to construct semantically equivalent documents which differ syntactically. This can be easily showed using an empty element, which can be both represented by `<element/>` or `<element></element>`. Therefore, equivalent semantics should map to equivalent syntactic values to be able to compute a meaningful hash value.

On the other hand, it is possible that within distinct documents, elements equal syntactically, but have a different semantic meaning. As an example, just consider namespaces: A change in the namespace declaration with the same prefix could change the semantic meaning of the elements using that prefix extremely, without changing them directly. To avoid this hazard, it must be assured that expanded names are used and that links are resolved to prevent equivalent hash values for different elements.

These problems have been recognized earlier, and several solutions have been proposed, for example DOMHash [22] and XML-Signature [9], where the latter generally uses CanonicalXML [2] for document normalization. Both ap-

proaches are robust against different encodings, namespace declarations, and attribute ordering. However, their aims differ.

DOMHash assigns a hash value to each XML element, where the hashes are computed recursively over the subsequent tree of that element. Hence, a change in one node affects all nodes on the path up to the root element. This allows an easy decision, whether a document was changed and if so, which part of the tree has been changed. The following example shows why this is not useful to our approach: A fingerprint contains an element as parent of a subtree probably. If any of its descendants would be updated – even outside the fingerprint – the hash of the parent element would have changed irrespective of the fact that the element itself did not change. Thus, the fingerprint could not be used as reliable indicator for the correct position of an edit operation. Furthermore, DOMHash does not allow the addressing of text nodes directly.

XML-Signature was designed to create a signature for a whole document or single parts of it. This approach has some major drawbacks for our scenario, too: First, each signature holds a substantial header. Considering that each fingerprint contains at least one hash value, whereas they usually contain more, the overhead seems to be significant. Second, the header must contain a path expression, which defines the signed part of the document or must contain the part itself. The latter is obviously useless when using hash values in a fingerprint for space saving reasons. The former is not very useful, either, because in our case, a fingerprint should be matched with its counterpart which has probably moved in terms of an absolute path.

Our approach is based on CanonicalXML [2], which is also recommended as a normalizing technique for XML-Signature, slightly extended towards the handling of text nodes, as already defined in Section 2.1. Thus, the pretty printed version of a document equals the in-line representation of it due to the disregard of text nodes consisting of white space exclusively.

For hash values computation, we use MD5 [23], which has been widely established as fast and reliable hashing algorithm with efficient implementations on most platforms. The different node types are hashed as follows:

- *Element nodes* are hashed on a concatenation of the expanded element name and their sorted attributes including their values.
- *Text nodes* are hashed on their unicode representation. Before hashing, leading and trailing whitespace is trimmed. This is a tribute to the behaviour of some pretty-printers, too.
- *Processing instructions* are hashed on their unicode representation.

One might argue that MD5 as a 128bit cryptographic hash function is too extensive for hashing single XML nodes which maybe have a smaller size. We plan to investigate for more suitable hash functions in the future.

By default, our approach hashes only the node itself. This behaviour is used for fingerprint generation. Moreover, it is also possible to hash complete subtrees in the style of DOMHash, which is used for conflict detection (see Section 4.2).

```
<?xml version="1.0" encoding="utf-8"?>
<delta>
    <insert path="/1/0" radius="2" digester="md5">
        <fingerprint>
            <hash dist="-2">
                -1263-6172-987-31-5624-45125-664-67-7258
            </hash>
            <hash dist="-1">
                75306-6037210511218-106-55-127-40107-62-47
            </hash>
            <hash dist="0">
                17-45-6-93-69-54-11709510561-1215453073
            </hash>
            <hash dist="1">
                96-8443-9-11-39113-540102-31744141-37
            </hash>
            <hash dist="2">
                17-45-6-93-69-54-11709510561-1215453073
            </hash>
        </fingerprint>
        <oldval>
        </oldval>
        <newval>
            <p style=T1>Lorem ipsum</p>
        </newval>
    </insert>
</delta>
```

Figure 4: An example delta containing just one insert operation.

3.3 Delta Format

At this point, we extend our definition of an edit operation (see Equation 4) by introducing the fingerprint defined above:

$$\text{operation} : (type, position, v, v', fingerprint(position)) \quad (8)$$

We stored the delta itself within an XML file, using the root node `<delta>`. Edit operations are mapped to the XML domain as follows:

- *type* maps to the element name of the edit operation and can be either `insert`, `delete`, or `update`. This node is the parent node of the rest of the edit operation.
- *position* is mapped to the attribute `pos`.
- *fingerprint* is stored in a subtree called `fingerprint`.
  - The attribute `radius` contains the radius of the fingerprint.
  - The attribute `digester` refers to the hash alorithm used (`md5` in our implementation).
  - The discrete values of the fingerprint are stored as text nodes as child of a `hash` element node with the attribute `dist`, which refers to the distance in relation to the anchor node.
- v is stored as child of `oldval`.
- v' is stored as child of `newval`.

Figure 4 shows an example delta[3].

[3]The different length of the hash values results from the fact that the hashes are stored as byte-array internally and only serialized in a string representation. The hash length is always 128bit.

Original edit operation	Inverted edit operation	Additional operation
(insert, *position*, $\emptyset$, v', *fingerprint*)	(delete, *position*, v', $\emptyset$, *fingerprint*)	
(delete, *position*, v, $\emptyset$, *fingerprint*)	(insert, *position*, $\emptyset$, v, *fingerprint*)	
(update, *position*, v, v', *fingerprint*)	(update, *position*, v', v, *fingerprint*)	*fingerprint*$[0] = \text{hash}(v')$

Table 1: Delta inversion is simple. Only update operations require to modify the fingerprint during inversion.

3.4 Inverting a Delta

As we store completed deltas in terms of [21], the inversion of a delta can be performed easily. Due to its definition, the fingerprint has just to be updated in case of update operations. Table 1 shows the rules for inverting edit operations.

4. APPLYING A DELTA

We contrast two cases for delta application. In the first one, the delta is applied to the document it was computed for. In the second one, however, the delta is applied to a different version of the document. For the first case (called standard case) we demand that every edit operation must be applied correctly. In the second case (called merge case) we could only promise a best-effort solution. Most of this section concentrates on the question how to deal with the merge case. Our solution for the standard case is described in Section 4.4.

Any change performed on a document can have a deep impact on the paths to the following nodes. A basic requirement for our patch function is to find the correct position of an edit operation, even if the address in the delta points to a wrong node in the meantime. As we do not want to perform an exhaustive search for that correct position, we define a neighborhood around the *designated point of operation* in which the correct delta position is supposed to be in Section 4.1. Conflicting edit operations could be identified, which is discussed in Section 4.2. Afterwards, we present a weight function to measure the quality of a potential position of an edit operation in Section 4.3. The patch procedure and its implementation is presented in Section 4.4.

4.1 Defining a Neighborhood

The merge case implies that the position of an edit operation potentially addresses a wrong node, as described before. In this case, our patch procedure has to search the correct node within the document to apply the edit operation. However, it is also possible that the node has been deleted or updated meanwhile, which prevents the patch procedure to identify the node accurately.

In both cases, we identify the best candidate node using a heuristic weight function described in Section 4.3. Using a complete approach, this weight function must compute the quality of all nodes within the document to find the best match. This exhaustive approach leads easily into a complexity trap. To avoid this, we define the *neighborhood* of the position of an edit operation as a set of nodes, which elements are regarded as *candidate* nodes for our weight function. The neighborhood has a radius ρ, with $\rho \geq 0$ and a size $m = 2\rho + 1$.

In our implementation, the neighborhood is computed according the rules shown in Figure 5, using XPath expressions. A so-defined neighborhood is composed only of nodes on the same hierarchy level. Figure 6 shows the neighborhood generation for the node `/1/0` of our example tree.

Note that these rules for neighborhood generation are preliminary rules, which are designed to fit many different document types. However, document type specific rule sets promise to be more appropriate, as the design could be tailored to the special characteristic of the document type. For example, for a spreadsheet document, a rule set is imaginable which would just look for matchings in the same column. An according interface for user-defined rule sets is planned.

```
takeNext(actNode, direction) {
    if (direction = left) {
        next := ./preceding-sibling
        if next = null {
            next:=../preceding-sibling::*/*[position()=last()]
        }
    } elsif (direction = right) {
        next := ./following-sibling
        if next = null {
            next := ../following-sibling::*/*[position()=1]
        }
    }
}
generateNeighborhood(anchor) {
    neighborhood[0] := anchor
    i := -1
    while abs(i) < ρ {
        neighborhood[i] := takeNext(neighborhood[i+1], left)
        i := i - 1
    }
    i := 1
    while i < ρ {
        neighborhood[i] := takeNext(neighborhood[i-1], right)
        i := i + 1
    }
}
```

Figure 5: The neighborhood of a *designated point of operation* is computed using XPath expressions.

4.2 Conflict Detection

Generally speaking, a conflict occurs when an edit operation tries to affect a node or subtree which has been changed in the meantime. Obviously, only update and delete operations can conflict, as insert operations do not act on a specific node. An insert operation can lead to a subtree inserted at a position if was no expect to, indeed. However, this is no conflict and will be denoted as false positive (see Section 5.1).

DEFINITION 2. *An update operation* op *conflicts, if for a candidate* p*, the anchor node of the fingerprint of* op *does not match the anchor node of* p*.*

It is possible to decide whether an update operation conflicts by just considering the fingerprint, as it just affects one node. For a delete operation, the subtree must be regarded, too.

DEFINITION 3. *A delete operation* op *conflicts, if for a candidate* p*, the anchor node of the fingerprint of* op *does not match the anchor node of* p*, or if* v *of* op *does not equal the corresponding subtree of* p*.*

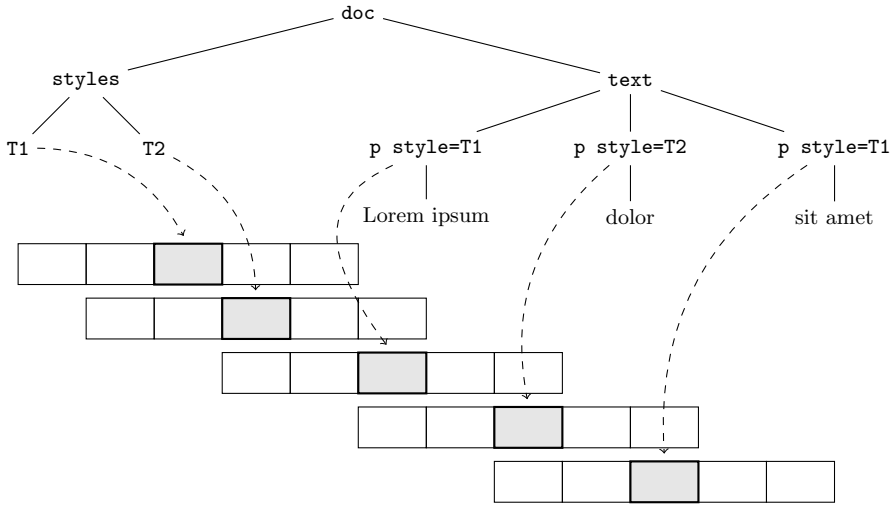

Figure 6: The neighborhood of the node `/1/0` with a radius $\rho = 2$ including the surrounding hash values.

Whether two subtrees equal, is decided on the basis of the recursively computed hash over v and v' described in Section 3.2.

In some cases, it might be useful to ignore conflicts, hence our patch procedure allows to ignore them.

4.3 Weighted Matching

Each edit operation will be performed only once during a patch run. Therefore, we need the ability to decide, which of the nodes in the neighborhood will likely be the correct point of operation.

We introduce a function which weights the quality of a candidate node with respect to the given fingerprint of that edit operation. It is not required that the fingerprint matches all of its counterpart nodes in the context of the candidate. The key idea of the weight function is that a fingerprint matching near to the anchor node is assigned a higher priority than a matching far away.

Before computing the fingerprint matchings, we must consider the type of the edit operation. For an insert operation, the anchor of the fingerprint points to the root element of the subtree to insert, which is obviously not part of the document to patch yet. Therefore, the anchor node can not match. To respect this fact, we define the sequence I, where r is the radius of the fingerprint.

$$I \stackrel{\text{def}}{=} \begin{cases} -r, .., -1, 1, .., r & \text{for insert operations} \\ -r, .., r & \text{otherwise} \end{cases} \tag{9}$$

The context of a candidate is computed in analogy to the rules presented for fingerprints defined in section 3.1.

A fingerprint matching is given, if the hash value of a node k_i in the context of the candidate p matches the according element of the fingerprint with the same distance i ($\text{dist}(p, k) = i$, $i \in I$):

$$match(k_i) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathit{fingerprint}[i] = \text{hash}(k_i) \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Thus, the quality of the candidate p is measured, depending on the amount of matching nodes in its context:

$$\mathit{MatchQuality}(p) = \frac{\sum_{i \in I} \frac{match(k_i)}{2^{|i|}}}{\sum_{i \in I} \frac{1}{2^{|i|}}} \tag{11}$$

The numerator counts all matches, whereas the denominator makes sure that the result is mapped to a value between 0 and 1. Table 2 shows example matching weights for update and delete operations.

As you can see, the distance to the anchor node has a strong influence on the weight (see Figure 7). Obviously, a fingerprint radius $r > 4$ would not contribute to the match quality in a noticable way.

The candidate with the best match quality is called *best candidate*.

distance i	quality if $match(k_i) = 0$
0	0.636
1	0.818
2	0.909
3	0.955

Table 2: The match quality for a fingerprint of an update operation with $r = 3$ if one node in distance i to the *anchor* does not match.

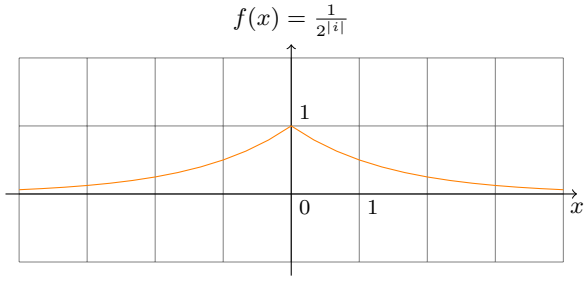

Figure 7: The influence of a node on the match quality decreases dramatically when gaining distance to the *anchor*.

4.4 Patching

The behavior of the presented patch procedure can be controlled by two parameters. The first one causes the patch procedure to skip the conflict detection. By using the second parameter, the patch procedure is given a *threshold value* which will be used as lower bound for accepting a computed *matchQuality*.

Our implementation works in two phases. First, all matchings are computed and stored in a accept- or reject-list. In the second phase, the matchings are either applied to the document or stored in a log file. The reasons for this separation in phases are as follows:

Any edit operations applied to the document potentially affect other edit operations. As the fingerprints are defined with respect to the original document, a second edit operation addressing a node near the first one could be prevented finding appropriate matchings – the fingerprint would not match as the document has been updated in the meanwhile. In addition, a re-labeling of nodes is extremely cost-intensive [14]. This re-labeling becomes necessary after an insert or delete operation to reconstruct a consistent node numbering. To avoid this task, our patch procedure stores a pointer to the in-memory representation of the node addressed by the matching instead of using its path in the accept-list. This allows a fast modification of the document tree in the second phase.

Up to now, we did not consider the question, what to do when a position in an edit operation does not refer to a node in the document to patch, except for the insert as last element of a subtree. First, we try to find the right most node in the subtree adressed by the edit operation, which would correspond to the XPath expression `../*[position()=last()]`. If no node is found, we repeat this recursively, following the hierarchy levels of the tree higher and higher, until we reach the root node of the document (which would mean that the document to patch is empty).

Our patch procedure also allows to ignore the fingerprints and to act like a "traditional" patch tool. This mode is obviously faster and can be used for a simple reconstruction of linear document versions.

The implementation was performed using the Java programming language and the XOM XML API[4].

5. EVALUATION

In order to evaluate our approach, OpenOffice text documents have been used. Changes have been performed on them, resulting in two different versions of each document, A', and A''. The deltas have been generated with a program extracting the operations which have been recorded using the "track changes" feature of OpenOffice. These deltas have been applied to the respectively other document version, as shown in Figure 1. The resulting documents have been compared with the expected merge result A^*.

The tests were performed on 6 documents with a size between 2 and 100 KB. The deltas consist of 1 to 10 edit operations, resulting in overall 60 edit operations.

5.1 Quality of the Result

To measure the quality of our patch procedure, we define following 4 categories for applied edit operations:

[4]http://www.xom.nu

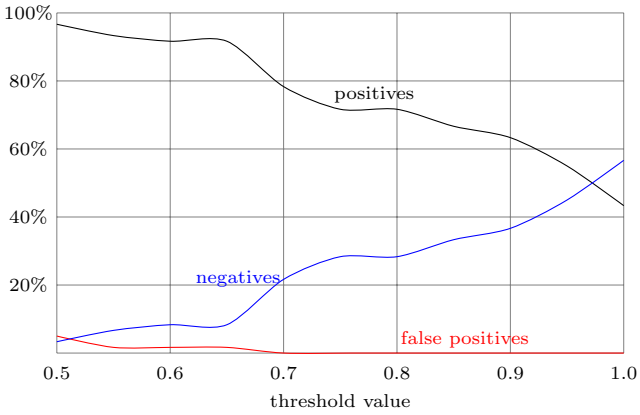

Figure 8: The test results show the high impact of the threshold value on the ratio of positives. No false positives could be found for a threshold ≥ 0.7.

- *positives* are edit operations, which have been applied to the delta.
- *false positives* are positives, which have been applied to a position where they were not supposed to.
- *negatives* are edit operations which have been rejected due to a match quality below the threshold value.
- *false negatives* are negatives, which have been rejected, even if a matching position would have been available.

Our test results are promising. The ratio of positives, false positives and negatives relating to all edit operations is shown in Figure 8. The ratio of false negatives is not indicated separately, as it almost equals the ratio of negatives. A "not false" negative indicates an edit operation which must not be applied, as a correct position does not exist anymore, thus meaning a conflict. In our test scenario, 5% of the edit operations match this criterion. All of them have already been found at a threshold of 0.5, therefore they are not indicated separately.

As a first conclusion, we state that a threshold value of 1 heavily decreases the amount of positives. Interestingly, no false positive occured using a threshold of 0.7 or higher. Between these both threshold values, the ratio of positives decreases significantly from 78% to 43%. Therefore, we recommend to use a threshold of 0.7 for a high reliabilty of the result without too much rejected edit operations.

If a high rate of positives is preferred, and few false positives are tolerated, we would consider a threshold of 0.55, where only 2% of the operations are false positives, with overall 93% positives.

5.2 Space and Time Complexity

Storing the context fingerprint of an edit operation implicates a higher space complexity of the resulting deltas. In our approach, this overhead is 0.6 KB per edit operation for a fingerprint radius of $r = 4$, and 0.5 KB for $r = 3$. This is significantly higher than XyDiff, which only requires an 0.2 KB overhead. To decrease the overhead of context fingerprints, one could pass on the ability to directly address the particular hash values of the fingerprint. By storing the

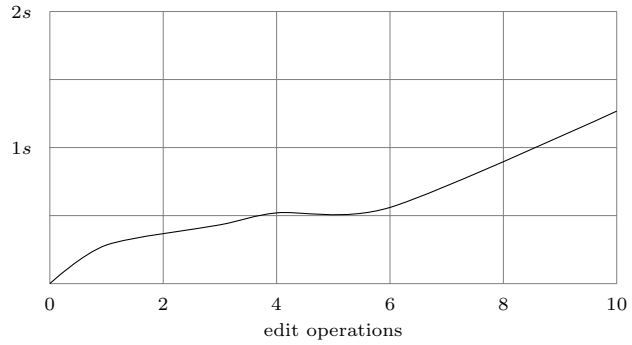

Figure 9: The performance of the patch procedure against the number of edit operations with a constant neighborhood radius of 10.

hashes sequentially, we are able to limit the overhead to 0.3 KB per edit operation, which is fairly in the range of traditional XML delta formats like XyDiff.

Figure 9 shows the performance of our implementation on a 2 GHz Pentium IV. Due to the restricted neighborhood (see Section 4.1), the runtime of the patch procedure only depends on the number of edit operations and the size of the document. Unfortunately, we can not perform a comparison of the execution time with other approaches, as they are not able to handle our test scenario (see Section 6). Still, our implementation could be improved. A speed-optimized version will be written and extensively tested.

6. RELATED WORK

The idea to respect the context of an edit operation is not new. In the domain of line-based deltas, this concept has proven its strength in the GNU diffutils [11, 13]. Its application to the XML domain and the use of hashing techniques however is new – no similar approach is known to us.

As already stated, only two XML diff/patch approaches can handle the complex task of finding all changes in two versions of one document efficiently, namely XyDiff [7] and faxma [20]. XyDiff identifies XML nodes using persistent identifiers, called XIDs [21], which are used to ensure the correctness of the diff and patch results. Consequently, XyDiff is not able to patch a document with a delta it was not computed for and aborts throwing an exception. The basic concept of XIDs hinders using XyDiff for document versioning and tracking outside the context of the Xyleme data warehouse, which for XyDiff was developed.

Faxma uses an own delta format, called XMLR [19]. Interestingly, the edit operations themselves are not addressed directly, but are embedded in a transformation script for constructing the new document version using the former one. This script works on absolute paths, however. Therefore, the correctness of the result can not be guaranteed. Absolute paths pointing to a no longer existing node lead to a program abort. This was the case in all of our tests (analogous to Section 5), except for two testruns.

Instead of applying a delta to a version of a document it was not computed for, a 3-way diff could be used to merge changes in the line-based domain [11], as well as in the XML domain [18]. These approaches are based on two internal 2-way diff runs – actually, the merges are computed by comparing the resulting edit sripts. In [11], a best-effort approach for merging is used, too, which assets and drawbacks are discussed in [13]. The delta model in [18] differs from ours, reverting changes is not possible.

Some state-based versioning approaches avoid the problem of addressing the edit operations by in-lining them into the updated document itself [10, 26], which requires storing and transmitting the complete document instead of just the deltas. Furthermore, embedding all version information might be unwanted due to privacy or security reasons, which is also the major drawback of systems using the operation-based approach [12].

Approaches exist to define edit operations on XML documents in a general way [3, 4, 28, 16]. However, these approaches use a database centric view towards XML documents and are not able to deal with documents updated in the meantime either, as they also use XPath expressions or similar languages for addressing nodes.

In the last years, an emerging community works on the idea of fingerprinting whole documents for similarity search. The main idea is to identify the similarity of documents by making use of hash collisions [15, 27]. This approach needs a minimum document size to be able to compute a meaningful fingerprint. As the context of edit operations is quite small, this technique is not promising for our scenario. Another approach tailored for XML documents was presented in [8]. As our fingerprint technique maps to linear data, thus dissolving the tree structure, this approach is not applicable, either.

7. CONCLUSIONS AND FURTHER WORK

This paper presents a new approach to handling edit operations on XML documents. Instead of enforcing a linear evolution of documents, our approach permits to merge different versions of a document.

Using the context of an edit operation, we are able to identify its correct position in the document to patch, even if it moved with respect to the position stored in the delta. This context is stored in a so-called fingerprint using hashing techniques.

The delta contains both the new and the old value of any edit operations. Therefore, our delta could also be used to revert changes. Furthermore, conflict detection becomes possible, which avoids the presented patch procedure to act on the wrong position by mistake. By defining a neighborhood, we restrict the search environment to possibly moved nodes, thus preventing an exhaustive search.

We present a weight function to measure the quality of a potential edit position with respect to the stored fingerprint. The accuracy of the patch can be controlled using a threshold value for the match quality.

Experimental results indicate that our approach generates reliable results. If possible errors are accepted by using a lower threshold value, the ratio of correctly applied edit operations increases strongly, whereas the drawback of false positives tends to be small.

During our research, some interesting questions emerged, which we plan to investigate in the future. First, we would like to perform tests using different hash functions, in particular fuzzy hash functions [27]. A major property of these hash functions is that their hash values are not affected by very small changes. The impact on real-world examples should be explored. Second, an API will be provided to in-

tegrate custom rule sets for neighborhood generation, which allows to tailor the neighborhood to the specific document type used. Finally, the usability of our patch program could be improved by a graphical user interface, which supports the user in resolving conflicts.

A precondition for a wide-spreaded use of our approach is the availability of good transformations from popular formats into the delta format we defined. Therefore, we will improve our implementation for the extraction of tracked changes out of ODF documents, which was used for our experiments. A transformation from deltas generated by XyDiff into our format is in progress. A corresponding transformation of faxma and possibly other approaches is to come.

Working with versions of XML documents is still not as handy as the line-based GNU diff and patch. Still, we have made progress in solving every-day problems in the handling of XML documents.

8. REFERENCES

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Oct. 1998.

[2] J. Boyer. Canonical XML version 1.0, 2001.

[3] E. Bruno, J. L. Maitre, and E. Murisasco. Extending xQuery with transformation operators. In *DocEng '03: Proceedings of the 2003 ACM symposium on Document engineering*, pages 1–8, New York, NY, USA, 2003. ACM.

[4] D. Chamberlin, D. Florescu, J. Melton, J. Robie, and J. Siméon. *XQuery Update Facility 1.0*, 2008.

[5] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD Conference*, pages 26–37, 1997.

[6] J. Clark and S. deRose. XML Path Language (XPath). Technical report, World Wide Web Consortium, 1999.

[7] G. Cobéna, S. Abiteboul, and A. Marian. Detecting Changes in XML Documents. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*, pages 41–52. IEEE Computer Society, 2002.

[8] D. de Brum Saccol, N. Edelweiss, R. de Matos Galante, and C. Zaniolo. XML version detection. In *DocEng '07: Proceedings of the 2007 ACM symposium on Document engineering*, pages 79–88, New York, NY, USA, 2007. ACM.

[9] D. Eastlake, J. Reagle, and D. Solo. XML-Signature syntax and processing, 2002.

[10] R. L. Fontaine. Merging XML files: a new approach providing intelligent merge of XML data sets. In *Proceedings of XML Europe 2002*, 2002.

[11] Free Software Foundation. *Comparing and Merging Files*, 2002.

[12] C.-L. Ignat and M. C. Norrie. Flexible collaboration over XML documents. In *CDVE*, pages 267–274, 2006.

[13] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3. In Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Dec. 2007.

[14] H.-K. Ko and S. Lee. An efficient scheme to completely avoid re-labeling in XML updates. In *WISE*, pages 259–264, 2006.

[15] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3(Supplement-1):91–97, 2006.

[16] F. Lam, N. Lam, and R. Wong. Efficient synchronization for mobile XML data. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 153–160, New York, NY, USA, 2002. ACM.

[17] E. Leonardi, S. S. Bhowmick, and S. K. Madria. Xandy: Detecting changes on large unordered XML documents using relational databases. In L. Zhou, B. C. Ooi, and X. Meng, editors, *DASFAA*, volume 3453 of *Lecture Notes in Computer Science*, pages 711–723. Springer, 2005.

[18] T. Lindholm. A three-way merge for XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10, New York, NY, USA, 2004. ACM.

[19] T. Lindholm, J. Kangasharju, and S. Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In V. Kumar, A. B. Zaslavsky, U. Çetintemel, and A. Labrinidis, editors, *MobiDE*, pages 49–56. ACM, 2005.

[20] T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 75–84, New York, NY, USA, 2006. ACM.

[21] A. Marian, S. Abiteboul, G. Cobéna, and L. Mignet. Change-centric management of versions in an XML warehouse. In *The VLDB Journal*, pages 581–590, 2001.

[22] H. Maruyama, K. Tamura, and N. Uramoto. Digest Values for DOM (DOMHASH), 2000.

[23] R. Rivest. The md5 message-digest algorithm, 1992.

[24] S. Rönnau and U. M. Borghoff. Intelligent merging of XML documents for distributed collaboration. In *Proceedings of the Distributed Intelligent Systems and Technologies Workshop*, pages 71–78, St. Petersburg, Russia, 2008.

[25] S. Rönnau, J. Scheffczyk, and U. M. Borghoff. Towards XML version control of office documents. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 10–19, New York, NY, USA, 2005. ACM.

[26] L. A. Rosado, A. P. Márquez, and J. M. Gil. Managing branch versioning in versioned/temporal XML documents. In D. Barbosa, A. Bonifati, Z. Bellahsene, E. Hunt, and R. Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2007.

[27] B. Stein. Fuzzy-fingerprints for text-based information retrieval. In *I-KNOW'05: Proceedings of the 5th International Conference on Knowledge Management*, pages 572–579. Journal of Universal Computer Science, 2005.

[28] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 413–424, New York, NY, USA, 2001. ACM.