Versioning and Merging XML-based documents
[COMP 790-063 Survey Paper]

Brendan Walters (waltersb@email.unc.edu)

December 18, 2009

Abstract

The three primary papers reviewed here are: Sebastian Rönnau and Uwe M. Borghoff's 2009 "Versioning XML-based ofce documents" [11]; Johnathan P. Munson and Prasun Dewan's 1994 "A Flexible Object Merging Framework" [9]; and Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma's 2006 "Fast and Simple XML Tree Differencing by Sequence Alignment" [6]. Various prior work and supporting papers are also employed and discussed.

An earlier version of this review focused on connecting the asynchronous collaboration capabilities of XML-based document merging to the pervasive, mixed synchronous/asynchronous spontaneous collaboration framework of Pering et al. [10], but this element has been largely dropped in order to expand on the concept of XML diff (not handled by Rönnau and Borghoff) and to refocus on contrasting the different approaches to document diff/merge described here. However, this analysis is well-framed by the concept of pervasive collaboration and I feel that the topic has some bearing on the concept of platform composition, so references have been retained in several contexts.

Also, algorithm performance is a significant focus of a number of the papers employed in this review, and as this is a matter of some importance for XML diff/merge tools given the current state of the art, it is discussed briefly in several places. However, detailed analyses of performance and comparisons to methods not covered here have been largely glossed over, again for the sake of brevity and focus.

1 Introduction

Most collaborative work is focused on the production of some sort of artifact, and many of these artifacts fall into the broad definition of "documents." Collaboration on document production tends to be hindered by the fact that most common document preparation tools are not initially designed to be collaborative, and in many cases cannot feasibly be altered to natively support collaborative work. Therefore a wide array of separate collaboration tools and platforms have been built to bridge the gap between various applications' provided functionality and that required for the specific types of collaboration they intend to enable.

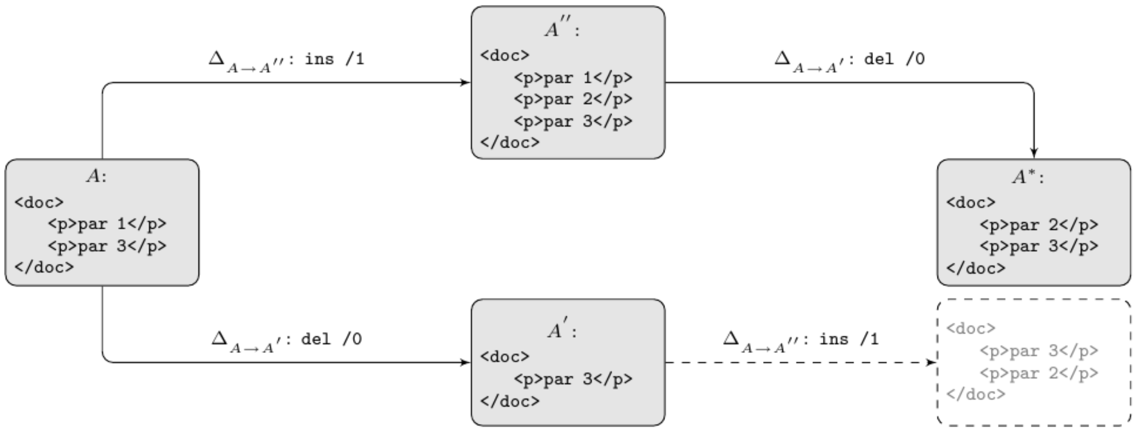
Given the usual necessity of operating completely outside of the the target application, it is perhaps unsurprising that some platforms seek to turn this separation into an advantage. One particularly thorough example is Pering et al.'s *Composition Framework* [10], which combines a wide variety of existing platform-specific tools into a middleware layer enabling *Pervasive Collaboration* - users can share files, screens, clipboards, and arbitrary applications in a manner transparent to the applications themselves, across different operating systems and device types, with minimal user direction. This powerful convergence of tools addresses many thorny issues of synchronous collaboration across disparate devices, but despite all the advantages of live interaction, there are some capabilities it lacks. With such a rich sharing environment, it is natural for users to work on documents on whatever device is handy even when disconnected, knowing that it can later be connected back to the group. Unfortunately, this introduces the need to support certain aspects of asynchronous collaboration in order to make a pervasive environment truly effective. (Incidentally, I feel that Pering et al.'s *Platform Composition* concept needs to include consideration of this element.)

The key problem introduced is the possibility of concurrent changes to unshared copies of a document, resulting in two separate versions with a common base. This problem can be avoided by preventing edits when not connected, or requiring that users obtain a *lock* on any file they wish to modify, preventing others from editing it until the first user is done and all copies are synchronized; however, this is severely limiting and cripples the concept of pervasive collaboration - if a user cannot decide to add to group work without being connected (to acquire a lock), it isn't really all that pervasive. Most collaborative file sharing systems (ie those that don't simply involve live application sharing, requiring live connections and scheduling) therefore employ some form of *optimistic replication* [15] - they allow modification while disconnected with the assumption that any conflicts thus generated can be resolved. Of course, applications can be built with this concern in mind and include capabilities to merge divergent files, but this is an unrealistic expectation for applications in general and would introduce potentially significant overhead. However, for certain types of asynchronously collaborative environments, some simplifying assumptions will often apply. For the case of frequently-disconnected mobile devices, T. Lindholm [3] points out that such devices typically process relatively simple data that often is or can be represented in XML form. The structured nature of XML raises the possibility of a (somewhat) general-capability *reconciliation engine* to perform structured data merges, possibly enhanced with format-specific semantic knowledge.

This paper addresses several approaches to merging of structured documents that could be used as part of a variety of collaboration systems. They are of course by nature asynchronous, but can be employed to allow editing by one user on multiple devices, or as noted above to make pervasively synchronous applications more pervasive.

2 The Document Merging Problem

Relatively simple diff tools have been used for quite some time for text files that tend to change on a line-by line basis, such as code in most programming languages. These tools align matching lines between the original and final versions of a file and produce listings of inserted, removed, and possibly changed lines (which may instead be represented by a delete-insert pair). Perhaps the most common is Unix's diff, the behavior of which some other tools copy; it simply finds the longest common subsequences between a pair of files and reports the rest as present in only one file (those in the older file can be considered "deletions" and the others "insertions"). The produced "diffs" are often employed to compactly represent a chain of file versions (as in version-control systems like CVS). When merging two sets of changes, the trivial solution of just getting two diffs and applying one on top of the other will in many cases produce different results depending on the order of application (Fig 1(a)), demonstrating that a more intelligent tool is required. Merging changes from two users is generally performed by a *three-way merge* tool, referring to the base (last common ancestor) file version and the two divergent new versions; these tools produce consistent output and usually deal with conflicts by notifying the user.



(a)

Figure 1: Simply applying the two diffs from conflicting versions gives inconsistent results [12].

2.1 A Brief Introduction to 3-way diff/merge

The Unix diff3 utility is one of the most common three-way merge tools and a reasonable example of the general behavior of such tools. The merge process for changed versions A and B of original document O is shown in Figure 2(a): first, it runs diff on (O,A) and (O,B) to obtain maximum-length matches (b), then it aligns the results on sequential segments that match on all three (c) to produce alternating *stable* (agreeing) and *unstable* (changed) chunks, and finally it applies changes in all conflicting chunks where only one of A or B has changed (d). This leaves some chunks that changed in bot A and B, for which diff3 indicates a conflict by printing all possibilities (e). Tools employing diff3 can apply internal rules to attempt to automatically resolve these conflicts or provide an interface for user resolution (e.g. KDiff3).

It seems intuitive that this will produce consistent and effective merges when changes are local and isolated - that is, no overlapping changes occurred, and changes are separated by large (for some definition of large) matching blocks - but it turns out that this is not always true [2]. In the example presented in Figure 3(a), an arbitrarily long sequence of “1,2” is edited in one case by prepending a new “1,2” and in another by replacing the final pair with “3.” The greedy matching lines up the new list start with the old one, resulting in a final chunk in which A appears to have suffixed a “1,2” pair rather than prepending it, which causes the final unstable chunk to finish as a conflict between the supposed addition in A and the replacement in B (Fig 3(b)). The example is somewhat contrived-looking, but it is simple for clarity and proves the point that simply ensuring that edits are small and well-separated is insufficient to ensure ideal resolution.

However, a slightly strengthened version of the isolation and locality concept above does result in consistent conflict-free merges; the separating sections need only contain a single unique line to assure this (and in practice, even without a completely unique line, alignment failures are rare). After our intuition is corrected by the example above, this makes intuitive sense - the problem in the above

$A = [1, 4, 5, 2, 3, 6]$
$O = [1, 2, 3, 4, 5, 6]$
$B = [1, 2, 4, 5, 3, 6]$

(a) inputs

A	1	4	5	2	3			6
O	1			2	3	4	5	6

O	1	2	3	4	5		6
B	1	2		4	5	3	6

(b) maximum matches

A	1	4,5	2	3	6
O	1		2	3,4,5	6
B	1		2	4,5,3	6

(c) `diff3` parse

A'	1	4,5	2	3	6
O'	1	4,5	2	3,4,5	6
B'	1	4,5	2	4,5,3	6

(d) calculated output

```
1
4
5
2
<<<<<<< A
3
||||||| O
3
4
5
=======
4
5
3
>>>>>>> B
6
```

(e) printed output

Figure 2: Normal operation of the Unix diff3 merge tool [2].

case is that the separator is merely a repeated series of the edit itself. The point has been adequately illustrated for our purposes, but a complete proof can be found in Khanna et al. 2007 ([2]), along with a thorough analysis of other intuitive and widely assumed properties of diff3 (e.g. idempotence, stability, and ability to handle certain bad cases), none of which it actually has.

Diff3 therefore tends to work well for plaintext files, which once established are usually edited in discrete sections separated by at least some unique lines; it also is widely used with much success for code, which is structured but is similarly edited in sections and tends to be kept locally consistent by individual edits, syntax-wise. However, it can introduce problems to structured data like code even with a perfect merge. As a trivial example, if one programmer changes the name of a variable in all instances within a function, but the other programmer adds a line to an otherwise unchanged sub-section of that function using the old name, the code will not compile (or worse, if names are merely swapped, a bug may be created and not noticed) without showing up as a merge conflict. The only way to protect against this and provide appropriate warning is to incorporate some knowledge about the structure of the data being merged.

This problem is exacerbated for document formats, which have complex structure that must be maintained for document consistency, and may even contain blobs of binary data (for example, included pictures). In order to address this problem, it is necessary to have a merge system that is aware of this structure and operates within it.

2.2 Object Frameworks

In 1994, Munson and Dewan proposed a general approach to the richly-structured-data merging problem centering on the application's model of the data artifacts that it processes. They introduced what they termed a *flexible object merging framework* designed not only to be capable of automatically merging, but also to offer a flexible system for specifying detailed merge policies [9]. They built a prototype on top of the Suite collaboration system, but the principles of their framework could be applied to any application using similarly structured objects. By building onto Suite, it was possible to make the merge framework application-independent (so long as applications used Suite objects for all document data).

O_1	$=$	$\emptyset$
O_2	$=$	$(1,2)^n$
O_3	$=$	$1,2$
A_1	$=$	$1,2$
B_3	$=$	3
A	$=$	A_1, O_2, O_3
O	$=$	O_1, O_2, O_3
B	$=$	O_1, O_2, B_3

(a) The setup: note that A and B each contain a single change, on opposite sides of an arbitrarily-long unchanged sequence.

"Correct" result			
A	$1,2$	$(1,2)^n$	$1,2$
O		$(1,2)^n$	$1,2$
B		$(1,2)^n$	3
	unstable	*stable*	*unstable*
O'	$1,2,(1,2)^n,3$		

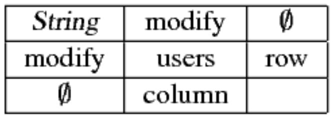
Actual result			
A		$1,2,(1,2)^{n-1}$	$1,2,1,2$
O		$(1,2)^n$	$1,2$
B		$(1,2)^n$	3
		stable	*conflict*

(b) The merge result.

Figure 3: It turns out that even local, well-isolated changes can cause diff3 to (erroneously) detect conflict (example from [2]).

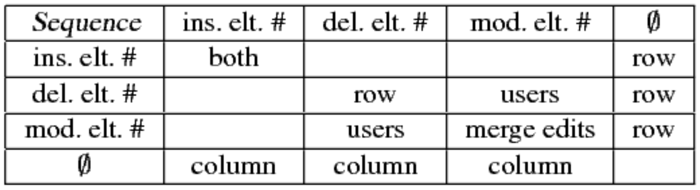
Because Suite is a collaboration framework with flexible coupling, it is specifically aware of changes made to objects as they are made (in order to synchronize them when coupled live), and can be made to store those changes into an edit history, or as tags to individual objects before they are synced. This natural change tracking takes the place of a full-document diff, saving time and guaranteeing accuracy of detected changes, both in location and value. Since edits can be reliably identified and attached to objects, the framework can perform merges by climbing the object hierarchy, and merge policy is defined at the object level. This handily sidesteps the structure-awareness problem experienced by simple diff tools above, by making the diff determination essentially part of the object model itself.

The core control of the object merging system is the *merge matrix*. The merge matrix defines the actions to be taken for every possible pair of actions from the "row" and "column" users. There are two fundamental object types - those that contain sub-elements (which may be objects themselves) and those that are atomic. Non-atomic objects are also given an atomic merge matrix with only the actions "modify" or null; this allows users to declare at some point during their collaboration that no further fine-grained merges on particular objects will occur, and that they will be managed as blocks. The atomic-object matrix is very simple: each user can only have modified the object or not (Fig 4(a)) (deletion or insertion of objects is handled in the context of the parent inside of which this action occurs). The simple example of a Sequence object (which contains sub-elements) in Figure 4(b) shows the main actions used: "row" and "col" mean to accept that user's edits only, "both" means to accept both (generally only applicable to insertions), "users" means to ask for user decision, and "merge" means to proceed to the rules for the child object in question. It is possible to specify custom merge functions as targets in any cell to enable more complex behavior, and specific object types may have other modification types than those shown. Merge functions can take into account the data in the merges, the time they were made, user identities, or any other data that a designer tells the system to encode. In this case, the behavior is to accept any modifications made if the other user made none, to add both elements if both users inserted new ones (in some order determined elsewhere), to delete an element deleted by both, but ask users for input if one user deleted an element that another modified, and finally to proceed to a lower-level rule set to merge changes on an element that both users have edited. This is a relatively simple, “safe” policy using only the default action types.



String	modify	∅
modify	users	row
∅	column	

(a) Merge matrix for an atomic object.



Sequence	ins. elt. #	del. elt. #	mod. elt. #	∅
ins. elt. #	both			row
del. elt. #		row	users	row
mod. elt. #		users	merge edits	row
∅	column	column	column	

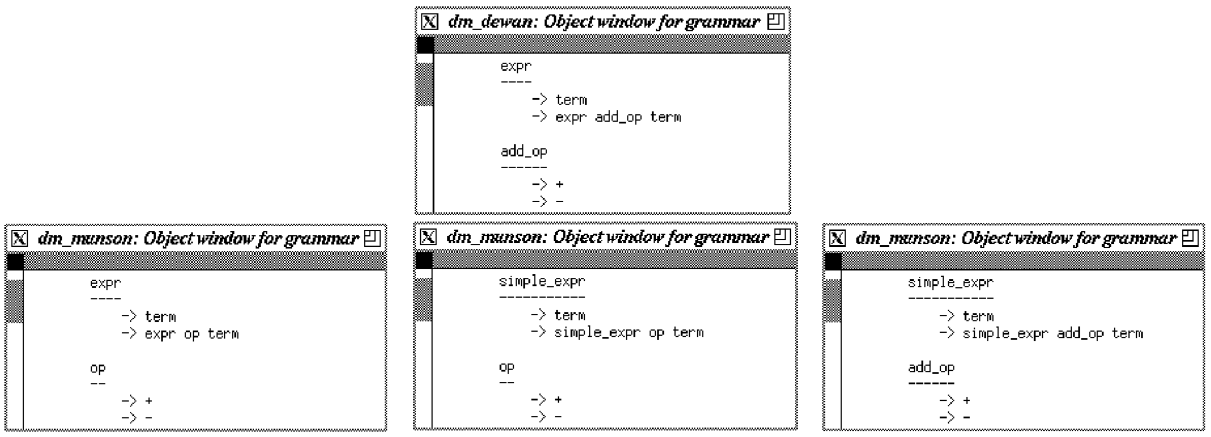
(b) Merge matrix for a Sequence object (containing sub-objects).

Figure 4: Merge matrices define actions to take for particular objects depending on what sort of changes (if any) have been made by the two users (row and column). [9].

Being completely tied to the structure of the data used (and customized per-application with appropriate merge matrices) enables successful automatic resolution in cases that would cause serious issues for simpler diff-merge systems. As an example, Munson and Dewan present a simple Yacc grammar editor. Figure 5(a) shows the original grammar, and Figure 5(b) shows the changes made by each user: Dewan has altered the operation in the second production of expr to add_op, while Munson has changed the name of expr (and its use in the same production). Because of the hierarchical processing and perfect change tracking, the merge system is able to see that while Munson changed the name of the object, it is the same one that Dewan was editing, so Dewan’s edits go into the correct object. Furthermore, when processing the modified production, the merge system proceeds to check the elements, and finding that no element is edited by both users, it can accept both changes automatically without conflict, giving the correct result in Figure 5(c). A line-based system could have failed to correctly align the changes, and a system without hierarchical processing would have had to ask for input on the multiply-changed production.

In operating directly on the document structure, this platform doesn’t merely sidestep the difficult problem of creating and aligning diffs; it also creates new capabilities unique to such a setup. Any other system would need to know something about (and therefore constrain) the data structure of the application to which it applied; this one is built into the building block objects themselves, allowing it to be simultaneously application-independent and universal as well as tailorable to any specific task. By building into an *existing* platform, this system can provide its unique guarantees to existing programs built with other capabilities in mind. Choosing a synchronous collaboration platform addresses the need discussed above for asynchronous capability to maintain smoothness, pervasiveness and convenience. More than that, it guarantees that apps are already following the one required constraint - they use only the provided objects to build their document data (and better yet, the system’s original functionality requires that it already have change monitoring capability).

Munson and Dewan discuss a variety of uses for their system, including complex function-driven merge policies such as “accept validated (eg spell-check passing) edits only” and policies to imitate tools such as diff3 or fileresolve, but as powerful and flexible as the system is, it has some important limitations. Most importantly, the prototype is limited to applications built on top of Suite, and the method itself to applications built on similar frameworks that have access to the data objects, can log all edits, and can reliably associate objects between versions. These are all natural in a collaboration framework, but one


dm_dewan: Object window for grammar

```
expr
----
    -> term
    -> expr add_op term

add_op
------
    -> +
    -> -
```

dm_munson: Object window for grammar

```
expr
----
    -> term
    -> expr op term

op
--
    -> +
    -> -
```

dm_munson: Object window for grammar

```
simple_expr
-----------
    -> term
    -> simple_expr op term

op
--
    -> +
    -> -
```

dm_munson: Object window for grammar

```
simple_expr
-----------
    -> term
    -> simple_expr add_op term

add_op
------
    -> +
    -> -
```

(a) Original (b) Changes (c) Result

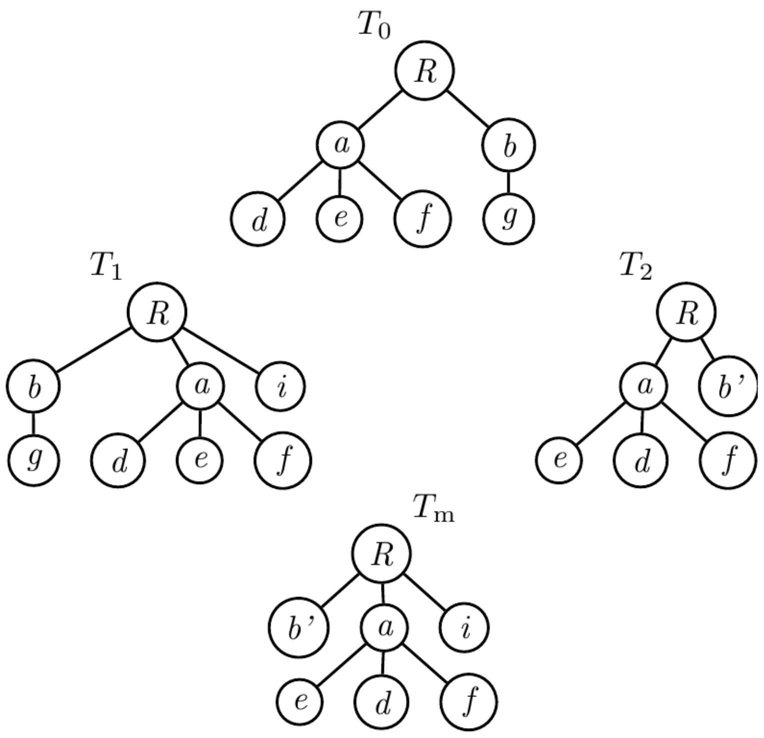
Figure 5: With intelligent handling of properties, changes to an object that itself has a name change are no problem. [9].

of the major issues discussed above is that few applications are built with collaboration toolsets. Thus, while the merge system itself is application-independent, it can only be used on an uncommon subset of applications. Ideally, a document merging system should be able to work with systems constrained not by architecture or toolkit, but by the material the merge system has to process - the format of the documents themselves. Fortunately for the potential generality of such a system, most major office suites are converging on several open standards all based on XML (primarily Open Document Format and Office Open XML, both of which are supported to some degree by OpenOffice/StarOffice/NeoOffice and Microsoft Office).

3 XML Diff

In addition to being a widespread standard and the basis for a growing number of document formats, XML is highly structured in a hierarchical tree. This means that some form of hierarchical processing can both identify matching elements to check for conflicts and appropriately place edits within the document. A simple conceptual demonstration in Figure 6(a) illustrates the intended behavior: changes to both the order and content of nodes are identified, and where they do not conflict, they are automatically applied. The example shown demonstrates that (provided with some means of identifying the changed version b' as a derivative of b), an XML document in which one user moves a particular element and another changes its contents can be automatically merged by applying both the structural and content changes. This is obviously something of which a normal line-based diff would be incapable (with no method to move changed lines to new locations), but doing it in the XML case, while possible, is less straightforward than in the object-based system above.

The primary reason why the Suite-object-based system above was so easy to describe is that a system uniquely tracking all objects and changes to them has no problem connecting differently-altered copies of any original object or figuring out exactly what those changes were. If instead we build a raw XML document reader, it can operate with any application that uses the chosen format, completely



(a) An example XML merge.

Figure 6: XML's tree structure assists in matching up moved elements and correctly merging structural and content changes [4].

independent of the application in a more meaningful way than the object-tracking system can be. There are two major approaches to this task: *operation-based* or *state-based* systems [12]. Operation-based systems hew closer to the object framework model above, employing detailed edit histories to identify the actions leading from the base to each current document state. This approach effectively allows the re-creation of the object model at any state, and by "re-playing" the edits, a merge procedure like that used in Munson and Dewan's system can be successfully employed. The state-based approach, on the other hand, relies only on the end state of the document.

The state-based approach has advantages even in the case of applications capable of supporting operation-based approaches. As Ronnau et al point out, "Persons (or organizations) often want to hide their editing process" [12]. More importantly, operation-based approaches require that the format support and always use very complete change tracking, both constraining the applicability of the method and imposing potentially signifiacnt overhead on documents. State-based approaches work with the final data format and only the final data format, giving them the greatest generality and, if comparable in results, the greatest capability.

In a state-based system, the first step toward merging documents is to intelligently create diffs for the two changed versions from their common ancestor. To do this, we need to understand the structure of our data. XML formats are naturally hierarchical, but the natural modeling of their data depends somewhat on the role of the format, particularly with respect to order: while even a database in XML necessarily has some ordering in the saved file, that ordering has no meaning. The general class of what could be called *document-oriented* XML, however, is naturally modeled as *ordered* trees [4]. Rather than being used for lookup queries, documents are presentation formats, meant to be consumed in an ordered (usually linear) fashion, and the data is stored in a corresponding manner. This means that not only must an XML diff system intelligently associate moved elements, it must also preserve information about relative positions and when merging, deal with changes not only to link structure, but to graph-irrelevant order among children of a node.

Lindholm et al. propose a method that "features a clean split between computing the XML difference and encoding it into an XML diff format" [6]. They point out that while some scorn has been directed at heuristic the approaches some have used to address performance concerns, there is no precise measure of a diff's "correctness" and optimality must be defined with respect to some cost function, and they note that for uses such as versioning, heuristic approaches have been favored due to their efficiency, including in early proposals by Rönnau [14], whose system is the topic of the next section. While operation-based systems have clear "correct" answers (the exact trail of user actions), a diff derived from the inital and final states alone might reflect any path between them and should be consistent (the "correct" matching is the actual edit path, which may be unrecoverable and is not useful as a diff). The important things are for the diff producer to be consistent and to attempt to optimize a cost such as diff length, calculation time, number if distinct actions in the diff, or any other metric useful to a merge system.

Noting that linear character-based diffs offer excellent compactness and performance, Lindholm et al. consider the possibility of employing such a system as an initial stage, thus reducing complexity in the first parser. Since XML is naturally tree-based, a method is required to transform that tree structure into one that, when analyzed as a sequence, will preserve important structural information. To that end, they employ XAS tokenization, separating XML into "events" including element start and end (ensuring splits on element boundaries), element contents, and attributes paired with values (ensuring that those atoms do not split) (see Fig 7(b)). Having chosen the encoding, the final element of the initial diff design is the available action set for changes between the original document d_0 and the new version d_1. Because sections of documents are frequently re-positioned during the editing process, the authors wished to support a *move* action. Given that the first stage was to be a simple sequence-alignment process, they avoided complex actions such as *copy* or *update*, limiting the system to a set that could be defined in terms of sequence matching only: *insert* (element in d_1 has no match in d_0), *delete* (element in d_0 has no match in d_1), and *move* (element in d_1 has a match in d_0). Note the absence of an "unchanged" or element - a move of no distance can encode this, and as insertions and deletions may change the address of unchanged nodes, this is a natural representation. (Updates are handled by insert-delete pairs, avoiding the need for the initial pass to detect partial matches within content nodes.) Figure 7(a) shows an example that involves an insertion (<i/>), a deletion (<e/>), a move (the <a>and <b> subtrees are swapped), and an update (root node <r> became <R>). Because all elements are treated as XML nodes in this example, only start and end element XAS events are used (SE and EE); single-tag elements are represented by adjacent start and end events.

The goal of the sequence-matching algorithm is to quickly find maximum-length matches and non-matches between the two documents, and in their 2006 paper, Lindholm et al. used a relatively simple greedy matching algorithm to verify the effectiveness of their method, pointing out that more complex existing methods could be employed as well. In order to detect moves, it is necessary to scan across the sequence repeatedly, unlike the simple greedy matching of Unix diff. To do this as quickly as possible, they employ a descending-size (from the list of sizes S = <48,32,16,8,4,2,1>) process with a rolling-

```
d0:  <r>
       <a>
         <c/><d/><e/>
       </a>
       <b>
         <f/>
       </b>
     </r>
```

```
d1:  <R>
       <b>
         <f/>
       </b>
       <a>
         <c/><d/>
       </a>
       <i/>
     </R>
```

Node	XAS Event
Element node e, start	SE(e)
Element node e, end	EE(e)
Text node t	C(t)
Attribute n with value v	AT(n, v)

(a) XAS event types.

(b) The example, containing all edit types.

Figure 7: Our XML diff example, and a table of XAS event types [6].

hash algorithm to rapidly check for matches. The documents d_1 and d_0 are sequenced into s_1 and s_0, which are then set as the initial values of **u** and **v** respectively, all elements in each being marked as ins(element) and marked with their original indices in s_x. For each size s starting at the largest in S, the remaining ins() elements in **u** are grouped into blocks s long, hashed, and matches are looked for in **v**, starting at the same index and sequentially checking in a "zig-zag" pattern, alternately moving outward before and after the matched index. Once found, matches are greedily expanded both forward and backward as far as possible, then replaced in **u** with cpy([start index in s_0], [end index in s_0 + 1]) and removed from **v**. Matches cannot span a previously-matched block (a gap in the s_0 indices in **v**). This procedure is repeated with successively smaller sizes, resulting in a final setup where **u** consists of ins() groups of inserted elements and cpy() groups of moved elements, while **v** consists of ins() groups of deleted elements. (For a complete pseudocode description, see [6], Fig.2; all code blocks have been left out of this paper for brevity.)

Figure 8(a) shows the XAS parse of the example in Figure 7(a), and Figure 8(b) shows the final steps of the matching process. Because the longest match is 5 elements long ($s_1^{5..10}$ to $s_0^{1..6}$), nothing can happen until the size reaches 4, and because sequential blocks in **u** are used (0..4, 5..8, 9..12), this does not find a match either. Once size reaches 2, $s_1^{2..4}$ finds a match in $s_0^{10..12}$, which then expands to $s_0^{10..13}$, and $s_1^{5..7}$ finds a match in $s_0^{1..3}$, which then expands to $s_0^{1..6}$. This gives the $\mathbf{u}_2$ result, and one more match is found at size 1 for the final $\mathbf{u}_1$ result, with deletions left in **v**.

$$\begin{aligned}
s_0 &= \langle {}^0\mathrm{SE}(r)\,, {}^1\mathrm{SE}(a)\,, {}^2\mathrm{SE}(c)\,, {}^3\mathrm{EE}(c)\,, {}^4\mathrm{SE}(d)\,,\\
&\quad {}^5\mathrm{EE}(d)\,, {}^6\mathrm{SE}(e)\,, {}^7\mathrm{EE}(e)\,, {}^8\mathrm{EE}(a)\,, {}^9\mathrm{SE}(b)\,,\\
&\quad {}^{10}\mathrm{SE}(f)\,, {}^{11}\mathrm{EE}(f)\,, {}^{12}\mathrm{EE}(b)\,, {}^{13}\mathrm{EE}(r)\rangle\\
s_1 &= \langle {}^0\mathrm{SE}(R)\,, {}^1\mathrm{SE}(b)\,, {}^2\mathrm{SE}(f)\,, {}^3\mathrm{EE}(f)\,, {}^4\mathrm{EE}(b)\,,\\
&\quad {}^5\mathrm{SE}(a)\,, {}^6\mathrm{SE}(c)\,, {}^7\mathrm{EE}(c)\,, {}^8\mathrm{SE}(d)\,, {}^9\mathrm{EE}(d)\,,\\
&\quad {}^{10}\mathrm{EE}(a)\,, {}^{11}\mathrm{SE}(i)\,, {}^{12}\mathrm{EE}(i)\,, {}^{13}\mathrm{EE}(R)\rangle
\end{aligned}$$

(a) The XAS parse of Ex 7(a).

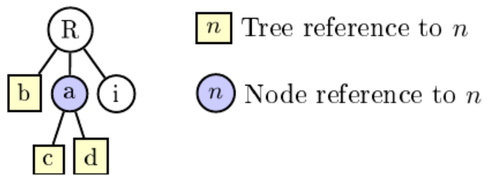
$$\begin{aligned}
\mathbf{u_2} &= \langle \mathrm{ins}(\langle \mathrm{SE}(R)\,, \mathrm{SE}(b)\rangle)\,, \mathrm{cpy}(10,13)\,, \mathrm{cpy}(1,6)\,,\\
&\quad \mathrm{ins}(\langle \mathrm{EE}(a)\,, \mathrm{SE}(i)\,, \mathrm{EE}(i)\,, \mathrm{EE}(R)\rangle)\rangle\\
\mathbf{u_1} &= \langle \mathrm{ins}(\langle \mathrm{SE}(R)\rangle)\,, \mathrm{cpy}(9,13)\,, \mathrm{cpy}(1,6)\,, \mathrm{cpy}(8,9)\,,\\
&\quad \mathrm{ins}(\langle \mathrm{SE}(i)\,, \mathrm{EE}(i)\,, \mathrm{EE}(R)\rangle)\rangle\\
\mathbf{v} &= \langle \mathrm{ins}(\langle {}^0\mathrm{SE}(r)\rangle)\,, \mathrm{ins}(\langle {}^6\mathrm{SE}(e)\,, {}^7\mathrm{EE}(e)\rangle)\,,\\
&\quad \mathrm{ins}(\langle {}^{13}\mathrm{EE}(r)\rangle)\rangle
\end{aligned}$$

(b) The resulting match lists.

Figure 8: The XAS parse and results of the initial matching process [6].

Having acquired the XAS sequence matches, we are faced with the problem that while the XAS events guarantee that matches align with element boundaries, they generally do not align with subtree boundaries. We need to assemble a tree model that aligns these matches to the actual document structure, and once again separation is helpful; the process is split into *match-to-tree* and *tree-to-diff* processes. In some sense, the result of the match-to-tree step, which the authors refer to as a *metadiff*, is much like the tagged object model of the previous approach, derived from the documents themselves. To encode this metadiff, the authors use a model they had previously developed called XML-with-references (XMLR) [5], while noting that other models including the W3C standard DOM could be applied. The XMLR language is a fairly simple extension of the base XML format, aimed specifically at versioning: it adds the *node reference tag* as a placeholder for a single node from another document (excluding its children), and the *tree reference tag* as a placeholder for an entire subtree from another document (including atomic nodes). The parse process operates on the original sequences s_1 and s_0, checking the final $\mathbf{u}$ and $\mathbf{v}$ for match values and ranges.

To parse the match lists into this tree format, it is necessary to read sequentially through the lists and only write out a node result when both the start and end tags of that node have been encountered; therefore a queue of tentative tree and node reference tags is maintained, changing tree references to node references when alterations are found. Start tags are checked for matches, and if matched, are checked for changes to attributes by looking for any attributes in the delete list). If they match and are unchanged, a node reference is added to the queue, and parsing proceeds. The node's children are then processed until the end tag is reached. The recursive processing of the children emits whenther changes have occurred, and for each node, when its end tag is reached the queue is printed if any changes occurred, but is discarded and replaced with a tree reference if none did. End tags are not information-bearing and are not checked for matches, being printed directly. (Again, the code listing is omitted for brevity, and can be found in [6], Fig.3.) The end result is an XML document matching the new version, with some nodes replaced by references, and some entire trees replaced by single tree reference nodes.



(a) The metadiff as a tree structure.

```
<R>
 <copy src="/0/1" run="1" />
 <node id="/0/0">
  <copy src="./0" run="2" />
 </node>
 <i />
</R>
```

(b) The final diff between d_0 and d_1, in terms of d_0.

Figure 9: The metadiff and final diff result for our example [6].

The metadiff now contains all the new information from d_1 and references to d_0, and need only be transformed into a useful format for quick processing. (Figure 9(a) shows the metadiff for our example as a tree). Nodes will be referenced by asolute numerical positions in (a subset of) the XPath format: "/0/2/1" refers to the root (0) node's third (2) child's second (1) child, and "./1" refers to the current node's second child. The diff is formed simply by printing the new elements directly, and outputting tree references as <copy src="[tree root ID]" run="#"/> and node references as <copy id="[node ID]"> (and accompanying </node>). The "run" attribute is used to make the diff more compact and clearer for human readers - it indicates that # successive nodes at this level are copied from d_1 (complete with children). Figure 9(b) shows the final diff in our example, illustrating the clear format matching the original (Fig 7(a)) and compactness. Employing the diff requires only parsing d_0, then printing the diff, replacing <copy> and <node> references with trees or single nodes from d_0. If desired, **v** can be included with the diff to show deletions, but it is not necessary for a simple diff (it will be useful for merges, however), and should be translated in much the same way to be most useful.

The authors show that their system (which they dub **faxma**) takes time approximately linear in document size and near-constant in edit count (with only slightly increasing variance with increasing edits). This is comparable (and on a large number of test sets, slightly superior to) the also-heuristic but slightly more complex **xydiff** approach, while it is significantly better than stricter models including **diffxml**, and **xmldiff**, and **diffmk**, which does not support the move operation. The simpler character-based **xdelta** is orders of magnitude better, but as explained above, not useful for XML versioning. Finally, as should be expected from the simple format above, the output diffs are approximately constant size over input sizes, and linear in edit count. While more complex actions such as copy could further reduce the output size, this method is fairly efficient and will in general be close to the difference in document size. The advantage in speed is significant; a somewhat optimized version achieved performance under 1 second for documents up to 1.6MB and very consistent linear scaling across varying document types, while two more-complex In contrast, **diffxml** and **xmldiff** both hit the 10 minute test limit before 1.6MB, **xmldiff** doing so before 1MB. Clearly, a greedy heuristic approach has benefits, and even with the speed show, the important move operation is supported, producig diffs that are both efficient and fairly compact.

4 XML Merge

Having an effective and informative XML diff is helpful, but not sufficient for merging. In a database application, it might be possible to ensure that elements retained the correct IDs, but in the ordered realm of documents, the absolute XPath notation used in diffs poses a problem: we cannot simply apply even completely correct, conflict-free diffs to versions for which they were not created. The dashed-line result in Figure 10(a) shows that a change to the position of list items will cause future value edits to be mis-applied, resulting in both inconsistency (depending on application order) and internal scrambling (not merely differences of order which could be potentially systematically made consistent). Clearly, a dynamic method of identifying the appropriate position for any edit is required, one robust to changes both of structure and of content.

It would be possible to solve the location problem by adding unique identifiers to nodes and somehow guaranteeing that they would not change when moved or when their content was changed, but in addition to being potentially quite difficult, this requires the cooperation of editing applications. One of the major goals of an XML merge system is to be able to merge documents from commonly-used office applications, and to even do so between applications, provided that they use a common format. Rönnau et al. proposed a system using *reliable context fingerprints* [11] to identify the correct location for changes so that they can be applied to edited versions. These fingerprints need to be calculated as part of the diff process, so their system requires new processes in both the diff and merge steps. As initially implemented, it employs an interesting blend of operation- and state-based approaches, although their goal is to enable purely state-based operation.

Rönnau et al. use an initial addressing system similar to that of Lindholm et al., a subset of XPath using only /node#/child#/... to refer to nodes in a fixed document. In this usage, "/1/0" refers to the the root node's (/) second child's (1/) first child (0). For this addressing, they ignore empty or whitespace-only text nodes in order to improve robustness in the face of variable methods of "pretty-

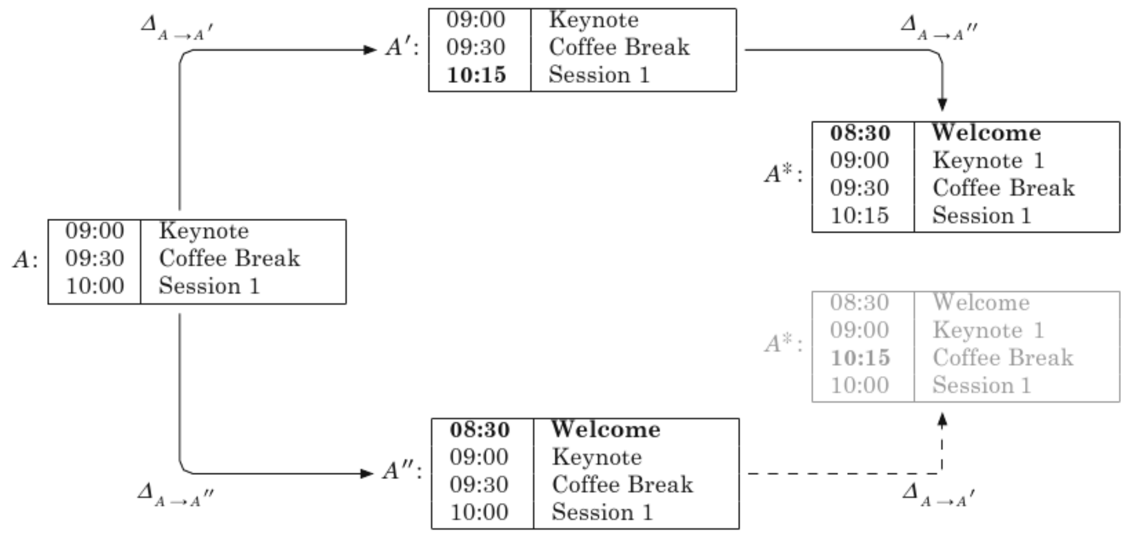


Figure 10: Applying diffs with absolute paths to documents for which they were not created will often produce scrambled results [11].

printing" output between applications or even different user preferences. Thus their merge results will largely preserve spacing from the base document. They then define the distance between nodes in terms of a linear walk of the document as a sequence: the distance between nodes i and j is the numer of nodes traversed (defined as encountering the start tag of the node) moving from one to the other (Fig 12(b) shows this in use). This has the effect that moving forward, nodes are ajacent to their first children, following siblings if childless, or the root of the next subtree if the rightmost leaf of a subtree. Thus the roots of document sections are closer to their content than to other sections (assuming hierarchy by sections, often the case of paragraphs at least), although closing sentences are close on one side to the following section root. This approach is both very simple (aquired by linear reading of the XML text) and likely to represent local structure well.

Rönnau et al. define the two sub-processes for their algorithm: the diff algorithm finds an edit script (or *delta*) $\Delta_{A\to A'} \in \Delta$ between two versions of one document $A, A' \in \mathbb{A}$, and the patch algorithm constructs a new document A' from A and a delta $\Delta_{A\to A'}$.

$$\text{diff}: \quad \mathbb{A} \times \mathbb{A} \to \Delta \tag{1}$$
$$\text{patch}: \quad \mathbb{A} \times \Delta \to \mathbb{A} \tag{2}$$

As diffs here are intended for use in merging, not to express the full form of the document as Lindholm et al.'s did, they need only contain operations for changes. This method uses only three edit operations: *insert*, *delete*, and *update*. Leaving out a *move* operation requires that such actions be represented as an *insert*-*delete* pair, but including *update* simplifies changing local content and separates the concepts of structural and content changes. Edit scripts are therefore sets of operations defined by *type*, old and new values v and v' (null in insert and delete, respectively), *position* (XPath-like), and $fingerprint$ (described below). The *insert* and *delete* operations act on entire subtrees, while the *update* operation acts only on the values in a node, not any children that may exist. The *position* value is the position in A at which an insertion takes place (shifting any children at that index or greater to the right), while delete and update refer to the address of the affected node (shifting following nodes left in the case of delete). Insertions and deletions are sequences of subtrees rather than single subtrees or single nodes so that only one of each will ever be applied at a particular address, avoiding ordering effects. The authors state that deltas can be easily inverted as shown in Figure 11(a) by swapping insert and delete operations, swapping old and new values for all ops, and in the case of updates, altering part of the fingerprint as described below; however, with the fixed positions alone, neither this inversion nor merging changes will work: inserts and deletes will affect the positions of existing nodes and cause edits to me mis-located.

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, $position, v', v, fingerprint$)	$fingerprint[0] = \text{hash}(v')$

Figure 11: The inversion process presented in [12]; it is defined in the same way in [12]. With fingerprints, this procedure will in most cases work as well as actually calculating the diff in the opposite direction; however, unmentioned in the paper is the fact that invert($\Delta_{A \to A'}$) and $\Delta_{A' \to A}$ will be far from equivalent.

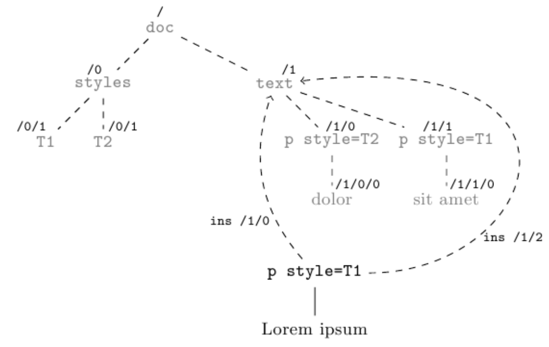
Ronnau et al. therefore require that all operations in a given delta be commutative, defined differently in their 2008 and 2009 papers:

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.*
(definition taken from [12])

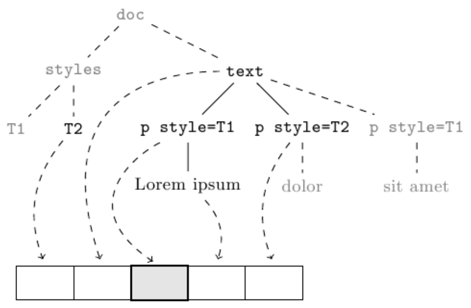
Definition 2 *Two edit operations $op_1, op_2 \in \Delta$, with $op_1 \neq op_2$ are commutative, if the position of op_1 does not point to an element of op_2 and vice versa.*
(definition taken from [11])

The first definition is more complete, covering the desired effects that all operations will work correctly independent of order (treating a delta as an unordered set) and of completeness (allowing some operations to be discarded, as in the case of merge conflicts). The second definition is defined in a manner calculable from the delta and should be equivalent, but with only the fixed positions, it requires a trick to be made to work: during a patch procedure, all edit operations are checked and stored in accept- and reject-lists rather than being immediately applied, so that each operation is presented with the same addresses and fingerprints. This makes deltas with A-based positions appropriate and consistent independent of application order when applying the delta to a document for which it was calculated. However, operations in two different deltas are not neccessarily commutative, so merging them will not work in the same way, and in the inversion case, the positions become incorrect; to address this, they introduce their *reliable fingerprints.*

Before discussing fingerprints, let us consider the simpler fingerprint-free operation of deltas. Both papers refer to the use of the deltas for fast *linear patching*, where they are applied to the document for which they are calculated, “just as described in the delta.” As mentioned above, this is achieved by storing all potentinal edits in an accept-list as they are processed, maintaining the structure and content of the base document for comparison. In order to avoid the cost (and order-imposition) of updating all node addresses in accepted edits when new operations change the tree structure, accepted edits are supplied with pointers to the objects they will edit, in memory. Presumably, inserts and deletes within a given node are sorted and performed end-to-start to maintain positin consistency. It is important to note that this procedure, which they state is fast because fingerprints can be ignored (although strangely they mention the possiblity of "ensuring correctness" by using conflict detection, something that should not be necessary if the delta works as advertised), will not work for inverted deltas. The positions in terms of A are retained when deltas are inverted, meaning that applying the inverted delta to A' will incorrectly locate edits. If the *position* values for insert operations are changed so that they point to the final position in the changed document A', the swap during inversion will result in still-correct position values for deletes and inserts; however, this breaks commutativity (application order will change which existing elements are shifted where, and which are deleted). Furthermore, there is no way to invert the position of update operations without analyzing the two tree structures. It seems therefore that fingerprints are indeed necessary for both merging and inversion (and are similarly imprecise in each case, since they are derived from A [which is required for maximum inter-diff consistency and to prevent errors in linear patching if employing the fingerprints]).



(a) The address structure for a document, and the addresses used to perform two different inserts: note that inserting as the last child employs an address beyond the end, and that inserting in either or both locations would use these same addresses - they depend only on the structure of A and not on the result of any other edits.



(b) The fingerprint (in a different document from (a)) of radius 2 for an update operation on node /1/0 (with the arrow pointing to the center, or anchor hash in the fingerprint). Note that the fingerprint for an insert or delete operation would differ, excluding the children of the anchor node.

Figure 12: The determination of positions and fingerprints for edit operations is soley from A; fingerprints are calculated from $A_{fingerprint} = A \setminus descendants(v)$, where v is the old value - this prevents deletions from referring to themselves; the anchor value is determined from head(v') for inserts, but is not used for matching.

Fingerprints are quite simple in concept; they are simply hashes of nodes within a particular radius surrounding edit operations, which can be used to align changes from one version to the altered document structure of another. Both node-only and subtree hashes are computed for this algorithm; however, the latter are only used for conflict detection. Node-only hashes must be robust against several potential sources of change: not only does XML permit semantically-equivalent statements of varying syntax, but content-irrelevant formatting such as user-customized pretty-printing may add or alter whitespace in varying ways. To maximize robustness against this, nodes are normalized according to the CanonicalXML guidelines with a few edits directed at text matching. All names are fully expanded (to avoid the effect of differing namespaces), whitespace-only nodes are omitted, and different node types are processed as follows:

- Element nodes are hashed on the concatenation of their name and sorted attributes with values.
- Text nodes are hashed on their unicode representation, with leading and trailing whitespace trimmed.
- Processing instructions are hashed on their unicode representation.
- Nonexistent nodes (at document boundaries) are given null hashes.

A variety of hash functions could be used - MD5 was used originally but it was switched to FNV for speed; the authors also discuss the possibility of augmenting this system with fuzzy hashes, allowing fingerprints robust against minor change (eg spelling corrections within a large text node). The adjacency of nodes is determined as described above, effectively matching the sequence of node start tags in the serialized document; however, different operation types use slightly different edits to this search. The fingerprint nodes are obtained from $A_{fingerprint} = A \setminus tail(v)$, where $tail(v)$ comprises all elements after the first node in the node, tree, or sequence in the operation's old value v. For updates, this is nothing, so all surrounding nodes are fingerprinted. For inserts, none of the nodes including the head node exist in A, so nothing is removed and the central (anchor) fingerprint hash is of head(v'), the root node of the first inserted tree. For deletes, all deleted elements except for the root node of the first deleted tree (the anchor node) are removed, producing a fingerprint of the context, not the deletion itself.

With the structure of the delta decided, a generation procedure must be found. It is here that the authors blend operation-based and state-based tactics - the prototype version does not compute deltas from the final structure of the edited XML documents, although it does use only the documents themselves. Instead, the ODF tracked changes feature is used, providing a guaranteed-correct account of the edit operations performed on the document without integration into the application itself. This could be viewed as a legitimate benefit of focusing on office documents, but the authors do intend to eventually create a final-state-based diff engine (probably xydiff-based according to their earlier paper, but they spent some time analyzing faxma in their more recent paper). Among other things, this will allow analysis of document types that do not track changes, including slideshow/presentation formats.

Even tracked changes guarantee correctness and map directly to the document, extraction of a commutative delta from the probably-not-commutative operation set is non-trivial. There are three particular sources of difficulty in doing this. First, ODF change tracking supports three operations: insert, delete, and format-change, the last of which does not preserve the original value and is not tied directly to the tree (being based on the document-header style list, and possibly being applied at a high level, but only appearing as a tag on the altered text nodes). Second, the change-tracking is at the character level, requiring remapping to the paragraph-as-text-node granularity of the ODF XML format. Third and most importantly, edits can span parts of adjacent subtrees, instead of being restricted to a subtree-bounded region with a single parent node. Changes are delimited by inserted start-change and end-change leaf nodes (or single nodes in the case of deletes, whose original data are stored in their document-header change list entries). Because only deletes store original values, only inserts and deletes are invertible, while format-change requires the determination of the original value from a root document (and import of all undefined styles for safety).

None of the papers published on this system go into great detail regarding the extraction process, but the summary is sufficiently clear. Insert and delete operations are initially split at paragraph boundaries. In cases where changes are contained within a paragraph (format and/or sub-paragraph insertion/deletion), they are combined into a single update operation. Similarly, any delete+insert pair operating on the same paragraph is replaced with an update. After all possible reductions to updates have been made, adjacent deletes and updates are glued into tree sequences (after expansion to full trees if necessary at boundaries). While updates are technically unnecessary, they are considered more informative, and are more useful for merging content changes to moved elements (especially with conflict detection).

Once a complete delta has been compiled, it can be applied either as a normal diff to A in the linear patching mode described above, or to a different version to produce a merged version A^*. Given that the application will necessarily be somewhat heuristic, a merge may differ depending on the direction of application ($\Delta_{A \to A'}$ applied to A'' versus $\Delta_{A \to A''}$ to A'), but the authors did not provide any test data for such inconsistencies. In the merge procedure, each operation searches for the best place to be applied, and to avoid the complexity trap of repeated exhaustive searches, the search is confined to a neighborhood of radius ρ. The neighborhood for searches is defined differently than that for fingerprints - by default, only nodes at the same hierarchy level are considered, on the assumption that these will represent the same type of content (and maximizing the search range for a given number of tests). The prototype implementation also supports alternate definitions for the neighborhood, and the authors note that some document types would require more complex rules (eg in a spreadsheet, looking for matches within columns or rows). Figure 13(a) shows such a search for the A' change applied to A'' from Fig 10(a), noting the hash matches and misses: first cells in each row match -1, +1 (both = cell) and -2 (table-row), while second cells miss -2 (cell != table-row), and all but the correct one miss 0 (10:00) and +2 (Session 1). I believe the figure to be slightly incorrect, as second cells in a row should have +1 = table-row != cell.

The example shown includes a perfect match, but in practice, this is not guaranteed, especially with a more-robust, wider fingerprint (the paper suggests using $r = 3$). Instead, a weighting function is used to choose the best match within the neighborhood (or to discard a change if no appropriate change is found). It gives a weight of 1 to the anchor node and 0.5^r for matches at each radius r.

$$I = \begin{cases} \{-r, ..., r\} \setminus 0 & \text{for insert operations} \\ -r, ..., r & \text{otherwise} \end{cases} \tag{3}$$

$$match(k_i) = \begin{cases} 1 & \text{if } fingerprint[i] = hash(k_i) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$quality(p) = \frac{\sum_{i \in I} \frac{match(k_i)}{2^{|i|}}}{\sum_{i \in I} \frac{1}{2^{|i|}}} \tag{5}$$

With this weighting, a fingerprint radius larger than their suggested $r = 3$ would have little effect, as is probably appropriate. A threshold for match failure can be applied to prevent the application of very-weak matches; Figure 14(a) shows the quality values for various single misses at $r = 3$. Figure 14(b) the result of tests at various thresholds on a set of 6 small documents with 1 to 10 edits each.

- *positives* are all edit operations that have been applied
- *false positives* are positives applied to the wrong location (or at all, if they should have been dropped)

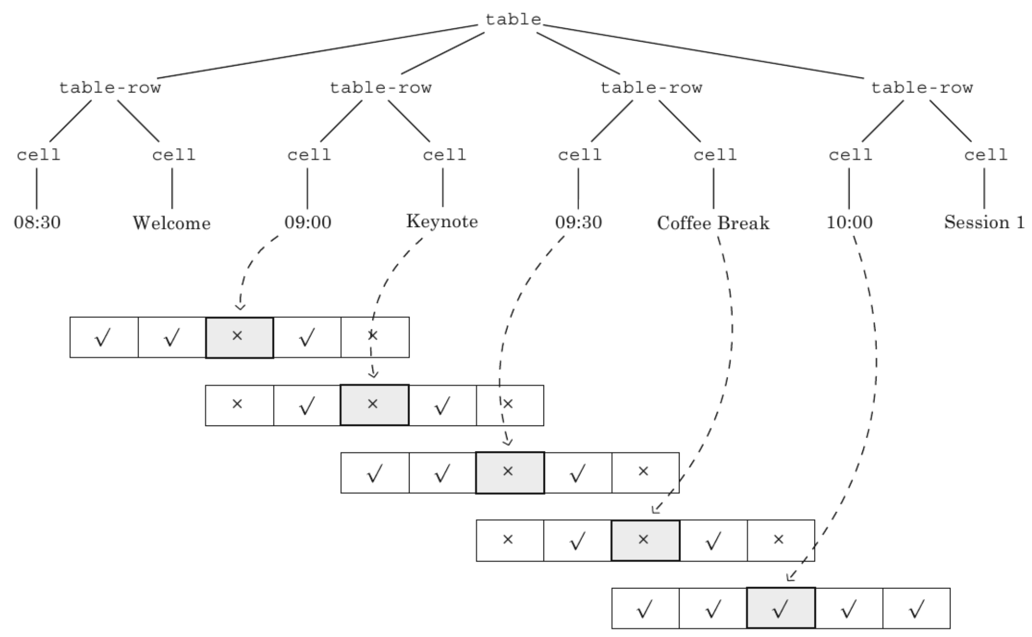


Figure 13: Searching for the best match for an edit to the node originally at /2/0/0, which has moved to /3/0/0. The fingerprint radius $r = 2$, and the neighborhood radius $\rho = 2$ as well (but recall that their adjacencies are defined differently). [11]

- *negatives* are all edit operations that have been rejected for a match quality below threshold
- *false negatives* are negatives that should not have been rejected, due to a proper location being available.

In the figure, false negatives are not shown because in the examples tested included it is almost the same as the number of negatives (few edits were applied in one document to an element deleted in another, the only possible cause of a correct negative). With the match quality on an anchor-node miss (the node to edit has been altered, but its neighbors have not, or the two immediate neighbors alone have changed) at .636, it is interesting to note that there is little change between 0.6 and 0.65 (the difference between being able to match to changed nodes and not), but the stronger drop at .55 is likely due to this change plus the effect of more distant nodes, which often change.

Given their early results, the authors suggest that a threshold around 0.7 would be a good choice to eliminate a large majority of misplaced edits (false positives) without sacrificing too many correct edits (78% at 0.7, 43% at 1.0); 0.55 proved to have very good accuracy, with 93% positives and only 2% false positives. Ideally, very low or zero false positives would not cause so many false negatives, and one approach to dealing with this is to introduce conflict detection, as supported by diff systems.

Conflict detection is a relatively straightforward addition, requiring only the ability to tell whether target nodes have been edited previously. Inserts cannot conflict, as they do not refer to existing nodes. They can insert at an unexpected location, but this is not detectable, nor is it strictly a conflict - only an error (possibly caused by other edits, but not interacting with them directly). Fortunately, the fingerprints contain all the information required to detect conflicts. Update operations conflict if

Distance i to the anchor	Quality for $match(k_i) = 0$
0	0.636
1	0.818
2	0.909
3	0.955

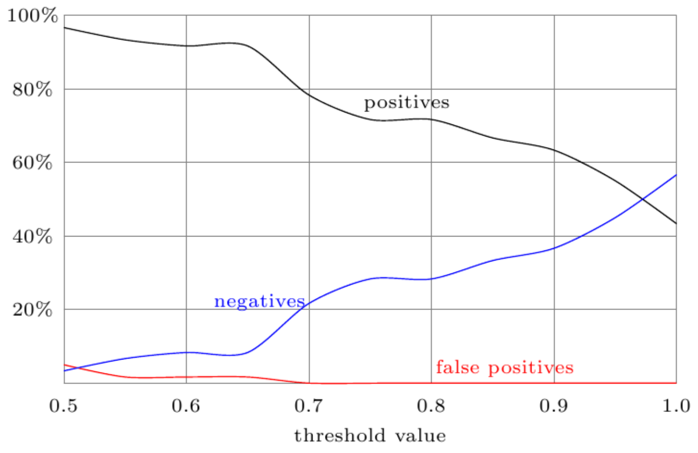


Figure 14: The values for single misses at various radii, and the effect of various thresholds on result quality. [11] & [12]

the anchor node of the update fingerprint does not match the anchor node of the candidate location at which it is applied. Delete operations conflict if the the anchor node fingerprints do not match as in an update conflict, OR if $tail(v)$ does not match the remaining nodes to be deleted; this operation will be sped up by the recursive subtree-hash calculation mentioned above, allowing a simple comparison of all the trees in the delete subtree sequence with the matching number of subtrees in the document to which it is being applied.

Figure 15(a) show the results from a later, larger test using search radius 6, fingerprint radius 4, and threshold 0.7. In this version, they have enabled conflict detection as well, which has significantly reduced the negative rate.

- *conflicts* are operations that have been rejected because of conflict detection (defined above).
- *false conflicts* are incorrectly identified conflicts, generally from cases where the wrong candidate node has the highest match quality.

They count correctly-identified conflicts as correctly handled, which seems appropriate given that there is no definite appropriate automatic response in those cases. The published system does not include an interactive mode for resolving conflicts, but the authors state that this is currently in progress - once completed, some studies of user response could lead to a refinement of conflict handling. With identified conflicts counted in, over 95% success with no false positives and $< 1\%$ false negatives seems to be quite successful. In fact, the 4.41% false conflict detection rate is the least severe error, as users will be notified and at least have a chance of noticing that the edit is in the wrong place; still it is inconvenient and possibly confusing to be asked to resolve a conflict that is in fact a mis-location.

Time performance was also good, with delta extraction scaling linearly with edit count (as expected) and remaining below one second for 700 edits, although rising at a much greater rate than faxma. somewhat surprisingly, the operation of merge was nearly constant-time in number of edits, probably because the primary cost is hashing, which is already done in the extraction step. Space overhead was significantly higher than with faxma, but still under 1KB for their example, so not a major concern.

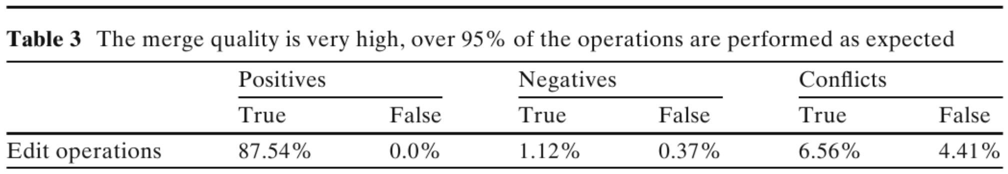


Table 3 The merge quality is very high, over 95% of the operations are performed as expected

	Positives		Negatives		Conflicts	
	True	False	True	False	True	False
Edit operations	87.54%	0.0%	1.12%	0.37%	6.56%	4.41%

Figure 15: The results for a more recent, larger test with a 50-page, 207KByte document, using 9- and 14-edit versions as bases and applying 8 to 288 further edits. [11]

These figures will of course change with the implementation of a purely state-based XML diff, but the core design seems effective.

The authors also discuss the probable impact of their choice of paragraphs as the finest-grained elements, pointing out that of this choice may increase conflicts, but reduces number of edits and therefore delta size (especially with tree-sequence glueing). They believe that in addtion to being more natural for the expected clustered changes (according to earlier work by Neuwirth), the reduction in reported conflicts would not likely improve the quality of the result. In support of this, they point out that edits to different portions of the same paragraph by different editors will frequently be inconsistent with each other (eg adding a clarifying sentence in two different places); presenting such actions as conflicts is probably safer, in addition to the benefits conveyed by coarser granularity. This is of course another element that will be more amenable to study once the user-interaction layer is completed.

While it is capable of operating only on output documents, the requirement that tracked changes be used is a definite limitation for Rönnau et al.'s system. As shown by Lindholm et al. and many other groups, developing a reliable, efficient XML diff is non-trivial, and will introduce errors of its own to compound with those of the merge process. Still, with the combination of the papers, it is clear that each is possible with significant success, and existing methods might be adapted to this purpose. The authors mentioned xydiff as a leading candidate in an early paper, but focused on faxma for comparison in the latest; however, they note that faxma's XMLR format is non-invertible, and doesn't map directly to their model.

5 Analysis

As briefly discussed in this paper, the claimed simple inversion of diffs is probably effective due to fingerprinting, but applying invert($\Delta_{A \to A'}$) to A' is still an application of the uncertain, heuristic merge process, likely in sufficiently complex cases to produce results different from the linear patch of $\Delta_{A' \to A}$ applied to A'. However, it is necessary to have such an invert capability because the current reliance on (not-completely-invertible) tracked changes means that there is no forward-generation process for $\Delta_{A' \to A}$. In fact, even with the introduction of a diff engine, this may be of value, because the diff will lack the guaranteed accuracy of tracked changes. In considering this, it is useful to note that there need be no distinction between a linear patch process and a merge - including the fingerprints of a perfect delta will not change the results because they will match exactly at the initial position checked, having been generated from there. In fact, even with an imperfect diff, the fingerprints will be generated from the positions found, so the perfect consistency with and without fingerprints will remain. It might be interesting to explore a re-calculation process for delta inversion, based on tagging locations to which edits are applied so that they can be checked for final position and fingerprinted in A' - this would provide a good method to check the results of diffs in opposite directions.

The move operation supported by faxma and other diffs is avoided in Rönnau et al.'s case maily because it would require two fingerprints - one for source and one for target. They state that this would reduce the probabilities during matching (which it would) and therefore confuse the use of the threshold. An unstated addition to this is that it would require better, diff-like analysis of the tracked changes to determine matches for moves. However, it seems that it would be wise to include, because separating a move into an insert and delete raises the possibility that one is applied and the other discarded. With a move, the element in question either remains in place or is reloacted; with an insert and a delete, it may also disappear entirely or become copied, the former of which could be a serious data loss. Fortunately, the most likely failure case in a merge of a move onto an edit is that a deletion fails and the insert does not, because the fingerprint has changed strongly at the original location but not the destination.

Still, it would be valuable to prevent even the move-to-copy failure, and while faxma is not fully merge-capable, the method it needs to detect moves contains the elements needed for similar capability in a merge system. Specifically, faxma is capable of separating structural and content change, and especially with the assitance of fingerprints, it should be possible to similarly detect blocks that have moved. Even without fuzzy hashes, moved trees could be fully fingerprinted (as nodes rather than recursively), and a weighting algorithm similar to the existing one could be applied to seek perfect or threshold-close moves, the latter being split into a move and an update. Of course, the move+update approach breaks commutativity, so to avoid more significant changes to the algorithm, the checks could use the existing recursive subtree hashes and be restricted to perfect moves and copies. The loss of approximate moves would not be too serious, as the usual case for that type of effect is in a merge, where one version moves elements and the other alters their value. At present, the existing system is equally effective in the case where the move comes first, and the mismatched delta containing the update is applied to it, finding the new location by fingerprint; however, the lack of a move operation means that an updated value cannot be relocated by a subsequently-applied diff with a delete-insert encoded move - this will insert the unchanged subtrees and delete the change in place, unless the delete fails because of the value mismatch (not too likely since the altered value is in the unfingerprinted child section). Conflict detection should at least catch the deletion of the edited node, BUT it will not be able to tell that this is part of a move, so it will be up to the queried user to notice that the value is changed and that it needs moving.

In both cases, the concept of inversion holds a possible method for improvement. In the case of faxma's diff, an inverse for a diff could be calculated, probably most easily from the matchlist (in principle, all of insert, delete, and move can be reversed, requiring only recalculation of parameters), and given that the input is always two documents, the forward diffs could be calculated in both directions and compared to each other's inverses. Any differences would signal areas worth examining further, possibly with special rules for reconciling differences or selecting "more-meaningful" changes. Given that the diff is the result of a deterministic greedy match and always produces the desire result, the difference in results can only be in compactness or some fuzzily-defined readability/meaningfulness metric. The likely difference would be in the expression of copies within changed/inserted sections: because the matching is greedy and chooses the first match, repeated elements may be connected differently when parsing from d_1 to d_0 as opposed to d_0 to d_1.

For the XML merge system, the inversion of potential value is that between $A_1^* = \text{merge}(A',\Delta_{A\to A''})$ and $A_2^* = \text{merge}(A'',\Delta_{A\to A'})$. THe merge process is presented with all of A, A', and A'', so it can extract both deltas and apply them to opposite changed versions; by trying the two application orders, a new kind of conflict could be introduced: sections that differ between A_1^* and A_2^*. Using either delta-element tags (indicating what tree elements were affected by what operations) or the new XML diff engine (needed anyway to detect the changes), the user could be presented with two options, rather than simple conflicts requiring manual resolution, and some false negatives could be caught, if they occur only in one version.

It is interesting that the tracked-changes processing had a significantly higher per-edit cost than the faxma diff, given that faxma also supports the discovery of moves. Presumably this cost is due to the hashing process, which may be less efficient than the near-constant-time rolling hash employed by Lindholm et al.. This may prove to be of significance with increasing document size (although 50 pages is fairly large, 207KB is fairly small), so an exploration of efficient hashes might be in order. Unfortunately, the use of fuzzy hashes is likely to involve an increase in cost - a compromise solution employing fuzzy diff on anchors only or conditionally in the case of poor or close match quality values might help.

Although it is not discussed here, Rönnau et al briefly present their GUI for user resolution of conflicts in [13]. Once their full system is complete, it will be interesting to compare the types of conflicts it generates compared with other diff systems, and to see if the implementation of a diff process led to the adoption of any of faxma's features.

6 Conclusion

There is little to add to the concept of a perfect-change-tracking, guaranteed-aligned system like Munson and Dewan's Object Merging Framework, beyond pointing out that it could be applied to newer collaboration frameworks like their Sync [8]. The concept of diffs and a merge procedure operating on a hierarchical document object model is exactly the basis for the XML diff and merge tools discussed (and most in general), but in most cases this requires derivation of a plausible set of edit operations from the state alone, and fitting by search rather than by recording. Even in the case of documents with change tracking, some post-processing is required.

While the methods described here strive for generality over XML documents, at least within one format family (for the merge system, any of the ODF formats supporting tracked changes), there is something to be said for application-specific tweaks. Tom Mens's 2002 survey of software merging [7] demonstrates that even that simpler-diff-friendly task permits many approaches, and also points out that language-specific syntactic and semanti knowledge can significantly enhance results. (The existence of detailed work on exactly that topic over a decade before ([1]) emphasizes the non-triviality of the problem!) Based on that evidence it seems very probable that the realm of documents could easily benefit from some semantic assumptions. Document content is of course much less strictly structured than program code, but certain behaviors could probably be encoded as custom rules. For example, in a text document, sentences inserted by different users within the same paragraph that are mostly the same are probably the same clarification, especially if parenthesized. Substantially similar paragraphs (especially with similar opening sentences) inserted in the same place should not be both applied (as Rönnau et al.'s merge system will), but should be presented as a posssible conflict. In a spreadsheet, insertions or deletions that cause column or row misalignments at the end of blocks should be viewed suspiciously. Some of these are harder to code than others, but some of the most useful are ones that avoid duplication as in the dual-paragraph-insertion case, and these are the simplest, especially with the help of a fuzzy hash. A system of context- or format-specific supplemental merge checks (in the format of Munson and Dewan's merge matrices with added conditions) could be inserted into these algorithms. In the XML diff case, a fuzzy hash plus contextual rules for what differences are acceptable could allow detection of moves with minor edits to the parent nodes themselves, or updates as such rather than as delete-insert pairs.

Finally, it would be interesting to explore an altered merge procedure for Rönnau et al.'s system that might be applicable to other diff types as well, including a slightly-remapped version of faxma's metadiff intermediate format. Because the delta application process relies on the commuativity of the delta, merging must be performed by applying one delta to another full version of the document, effectively applying the two deltas separately and consecutively. However, that commutativity requirement could be employed to direct a delta-merging procedure. The operations in $\Delta_{A \to A''}$, added one at a time to $\Delta_{A \to A'}$, could be checked for conflict and resolved:

- Intersecting inserts would be resolved by merging the tree sequences, with a consistent procedure to determine which comes first in the case of no overlap, and with overlapping subtrees used for alignment (and not duplicated).
- Matched deletes would be eliminated; since they are both based on A, they will match perfectly, avoiding one possible data-loss source (false positive match of a repeated delete) entirely.
- A delete on an edited node will be moved to a separate lists, the conflict deltas.
- Overlapping updates will also move to a conflict delta.
- If moves are implemented (addressing the order dependence for moved segments with edited elements described above), then updates inside of moves will require special handling, but they can also be detected without move support from updates that match deletes. Instead of moving to a conflict delta, if an update matches a delete but there is an insert elsewhere of the relevant subsection of the delete, the insert will be edited to reflect the update.

The final merged list could then be applied as a single diff, and would produce the same results every time. In addition to removing the current order-dependence for mixed structure and content changes (one user moves what another edited), it allows detection of those events as edit+moves without requiring explicit move support. As with any change, this would have a cost, specifically in significant merge time; however it could be added on without any changes to the operation of the system in other modes.

A final ability that could be added to any of these modular-procedure systems is the cabability of producing (and merging) forward diffs from reduced versions of documents onto the full versions. Linholm's suggestion of an XML merge to reconcile edits made on mobile devices to the original, richer form of a document suggests this use [4]. It should be straightforward to calculate the diff from the reduced-for-mobile-devices version to the final edited version there, then apply that diff to the whole document - this will avoid generating delete events for all the removed detail. Fingerprints shoud be consistent as they are mostly content-caused (Rönnau et al.'s removal of device-specific formatting will help), and they could be customized to this application. Relatedly but in the other direction, a pair of edited versions could be compared against each other to get diffs in either direction, and an intelligent inversion of the resulting inserts and updates could produce a reasonable diff to a possible shared base version.

Overall, general diff systems are quite powerful and if they are tied to an object model, they can be enriched with a variety of semantic knowledge for further effect. Frequently, limiting properties of such systems may be turned to advantage, as in tha case of the commutativity requirement, because those properties require that the system have the ability to check or enforce them. While the area of automatic XML merging is young, the array of possiblities for extension to just these few systems suggests that there may be a proliferation in the near future just as there is already beginning to be in XML diffs and has been in software merging. Perhaps there will even be some analytical value to diff systems that can learn about a user from the inter-version changes he makes to his documents; the horrible prospect of a user-habit-trained Clippy II springs to mind, but hopefully someone will come up with a less frightening use.

References

[1] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(3):345–387, 1989.

[2] S. Khanna, K. Kunal, and B.C. Pierce. A formal investigation of diff3. *Lecture Notes in Computer Science*, 4855:485, 2007.

[3] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97. ACM New York, NY, USA, 2003.

[4] T. Lindholm. A three-way merge for XML documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10. ACM New York, NY, USA, 2004.

[5] T. Lindholm, J. Kangasharju, and S. Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In *Proceedings of the 4th ACM international workshop on Data engineering for wireless and mobile access*, pages 49–56. ACM New York, NY, USA, 2005.

[6] T. Lindholm, J. Kangasharju, and S. Tarkoma. Fast and simple XML tree differencing by sequence alignment. In *Proceedings of the 2006 ACM symposium on Document engineering*, page 84. ACM, 2006.

[7] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, pages 449–462, 2002.

[8] J. Munson and P. Dewan. Sync: A system for mobile collaborative applications. *IEEE Computer*, 30(6):59–66, 1997.

[9] J.P. Munson and P. Dewan. A flexible object merging framework. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 231–242. ACM New York, NY, USA, 1994.

[10] T. Pering, R. Want, B. Rosario, S. Sud, and K. Lyons. Enabling pervasive collaboration with platform composition. *Proceedings of Perviasive09*, 2009.

[11] S. Rönnau and U.M. Borghoff. Versioning XML-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, 2009.

[12] S. Rönnau, C. Pauli, and U.M. Borghoff. Merging changes in XML documents using reliable context fingerprints. 2008.

[13] S. Rönnau, G. Philipp, and U.M. Borghoff. Efficient and reliable merging of XML documents. In *Proceeding of the 18th ACM conference on Information and knowledge management*, pages 2105–2106. ACM, 2009.

[14] S. Rönnau, J. Scheffczyk, and U.M. Borghoff. Towards XML version control of office documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, page 19. ACM, 2005.

[15] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005.