Comput Sci Res Dev (2019) 34:3–16
DOI 10.1007/s00450-013-0253-5

REGULAR PAPER

Using versioned trees, change detection and node identity for three-way XML merging

Cheng Thao · Ethan V. Munson

Received: 12 June 2013 / Accepted: 26 November 2013 / Published online: 29 November 2014

Abstract XML has become the standard document representation for many popular tools in various domains. When multiple authors collaborate to produce a document, they must be able to work in parallel and periodically merge their efforts into a single work. While there exist a small number of three-way XML merging tools, their performance could be improved in several areas. We present a three-way XML merge algorithm that is faster, uses less memory and is more precise than previous algorithms. It uses a specialized versioning tree data structure that supports node identity and change detection. The algorithm applies the traditional three-way merge found in GNU *diff3* to the children of changed nodes. The editing operations it supports are addition, deletion, update, and move. The algorithm is evaluated by comparing its performance to that of the previous algorithms, using synthetically generated XML documents of a range of sizes and modified by varying numbers of random editing operations. The prototype merge tool used in these tests also includes a simple graphical interface for visualizing and resolving conflicts.

Keywords Three-way merge · Collaborative editing · Versioning system · Algorithm · XML · Data structures

C. Thao
Department of Mathematical and Computer Sciences,
University of Wisconsin-Whitewater,
Whitewater, WI 53190-1790, USA
e-mail: thaoc@uww.edu

E. V. Munson (✉)
Computer Science, University of Wisconsin-Milwaukee,
Milwaukee, WI 53201-0784, USA
e-mail: munson@uwm.edu

1 Introduction

In this article, we describe and evaluate a new algorithm for three-way merging of XML documents. This algorithm is important because three-way merging is a critical element in any modern version control framework. The algorithm is novel because it is substantially faster and uses less memory than previous three-way XML merging algorithms. We show this through an empirical evaluation on simulated XML files, comparing our algorithm directly to the previously published alternatives.

The key difference between our algorithm and the alternatives is our use of *node identity* to keep track of which parts of an XML document have changed. We assume that each time an XML file is saved, all newly created elements become marked with a unique identifier and that these unique identifiers are not changed by future editing sessions, no matter what other changes are made to the element. This use of UIDs contrasts with the approximate matching approaches used by other XML merging algorithms and is key to the higher efficiency of our algorithm.

The remainder of this section will motivate the three-way XML merging problem, briefly discuss conventional version control systems, explain what makes three-way XML merging special and introduce notation used in the paper.

1.1 Motivation

Research on software engineering tools has focused on tools for program source code and the many other formal languages that are central to system development. But it is well known that many important software engineering artifacts are natural language documents many of which are simply conventional office documents (requirements and design narratives, user documentation, high-level testing plans, meeting

transcripts and notes). Others are structured, but not so formal as source code (e.g. various UML diagrams, figures showing architecture, etc.). Like source code, these documents are produced by collaborative teams and they evolve in the same ways. So, they need the same version control services that are used with source code: branching with some means of tracking ownership and responsibility, differencing to understand changes, and merging to create a single shared version from multiple branches.

Today, non-source-code documents are usually produced using modern GUI applications that save their files in an XML syntax based on either open or proprietary XML document structures. For example, files for Microsoft Office use the Office Open XML Format, while OpenOffice and LibreOffice use the Open Office XML format. In both cases, the "saved file" is actually a compressed archive containing multiple XML files. Since all of these office suites now support change tracking and merging of versions, it might be argued that there is no room for innovation in version control of natural language documents. But we are interested in making it easy for any XML application to support a variety of useful versioning interfaces via standardized XML versioning. Furthermore, it is clear that office applications do not yet provide adequate version support because collaborating authors find that they must keep copies of many different versions of their documents and to construct cumbersome naming schemes to distinguish those versions from each other.

Version control of XML files presents challenges and opportunities not seen with simple text files.

The challenges come from the fact that XML files often lack the line breaks that function as structure boundaries in text files. Large portions of the document content may have been manually entered, but the start and end tags that define the document's tree structure and carry the attribute values are generated by software that has little need for line breaks. For example, both Microsoft Office and OpenOffice produce XML files with exactly two lines. The first line is short and contains the XML declaration. The second line contains everything else, and in the files that hold the main document content, may be many kilobytes long. Thus, line breaks have little value for version control in XML-based applications.

It is the tagged tree structure of XML that provides opportunities because it provides a rich syntactic structure that can be mapped to semantics when document schemas are well-designed. Good tools could use this structure to find versioning information that has much greater utility for human users than the simple line-based differences of SCM versioning or the edit-operation based change model used by office suites. It is worth noting that the Office Open XML and Open Office XML representations are very presentation oriented and do not have particularly good semantics, but they do clearly mark paragraph boundaries and divide tables into cells. Other XML formats, such as those for e-books [1] and for UML diagrams [2], are more promising.

XML provides another advantage over simple text through its namespace mechanism. XML namespaces are designed to allow third-party applications (TPAs) to add elements or attribute values that are marked with a TPA-specific namespace. The TPA can use these to embed extra content (via elements) or to annotate existing content (via attributes), provided that the main application is not disrupted by the additional material and preserves that material through the load-edit-save cycle. The research presented here takes advantage of this mechanism by attaching UIDs to document elements using attributes. This allows our versioning software to reliably distinguish between *move* operations and the combination of a *delete* followed by an *insert* of the same or similar material at a different location. For example, if source code is stored in the SrcML format [3], our software knows the difference between these two cases:

1. A Java method is removed from position A and a different method is inserted in position B.
2. A Java method is moved from position A to position B and is then renamed and substantially modified.

1.2 Conventional version control systems

Beginning with the introduction of the SCCS [4] and RCS [5] systems, version control software has become a standard element of modern software development practice. SCCS and RCS focused on managing versions of individual files and did so on conventional file systems. The introduction of CVS [6] extended RCS to better support complete projects and allow remote access to version repositories. Recently, Subversion [7], Mercurial [8] and Git [9] have been introduced, each providing somewhat different approaches to distributed version control. Subversion uses a centralized version repository and adopts the "product versioning" model, in which the version history is connected with the entire project, rather than individual files having separate histories. Mercurial and Git use replicated local repositories and focus on high performance during merging.

All of these systems have a clear focus on supporting software development and have adopted some limiting assumptions. None of them attempts to understand the semantics of the files that are versioned, probably because they must support a variety of languages with very different grammatical structures (e.g. lex, perl, Ant, and Java). Files are either simple text, where line breaks are considered to have utility for information organization, or are binary files that lack meaningful internal structure. The focus on simple text makes sense for software developers, who must work with a variety of formal languages and who are also sophisticated users of low-level textual search and analysis tools (e.g.

grep, diff, etc.). But more and more, software developers are using advanced editing tools with graphical user interfaces that could easily support more complicated representations built on top of XML or other structured document languages. And most other "knowledge workers" are already using tools whose underlying representation is an XML document type with much richer syntax and semantics than raw text.

Three-way merging is an important operation in version control because developers' independent changes must, at some point, be merged into a common result. For text files, there exist tools like GNU *diff3* [10] for merging two documents derived from a common base document. This process is called *three-way merging* because it requires three documents: the original or *base* document and the two modified or *derived* documents. Software configuration management (SCM) tools such as CVS [6] and Subversion [7] make heavy use of three-way merging to support collaborative editing of program source code.

1.3 Three-way merging for XML

Three-way merging for XML documents is considered to be meaningfully different from text-based merging for several reasons. First, where a text file is considered to be a *sequence of lines*, an XML document is considered to be a *tree of attributed elements*. Second, XML has non-trivial semantics for differences.

The XML language definition specifies that the order of elements is significant, but that the order of attributes in an element's start tag is not significant. So, XML defines *ordered trees* of elements with *unordered attributes* on each element. The ordered tree semantic is obviously important for text documents such as OpenOffice XML and it also matches the standard semantics of the painter's algorithm for two-dimensional graphics documents, as in SVG. However, many XML document types have unordered semantics for at least some elements of their trees. For example, when defining linear gradients in SVG, the order of the `stop` elements is not important.

Although one could build a specialized XML merge tool for a specific XML document type, this paper addresses general XML documents and assumes that element order is important and attribute order is not important. This influences how the algorithm detects conflicts, especially insertion and move conflicts. The proposed algorithm also assumes that there are unique identifiers at least for all base document nodes. This includes those elements in the modified documents that were originally part of the base document, even if they have been substantially modified in the intervening versions. We use the attribute `mouid` to store the unique ID of an element.

There exist both commercial and research tools to perform three-way merging of XML documents [11,12]. They differ from *diff3* in that they merge *trees* of nodes that have both attribute values and textual content rather than sequences of text lines. For efficiency reasons when matching nodes in different versions, these tools assign hash values to the nodes and the node content. While the use of hash values speeds the matching process, it leaves the tools unable to identify matches between versions of XML document nodes that have undergone large transformations. Performance remains a problem, because to do a complete merge, these tools build three, if not four, complete document trees in memory even when changes are trivial. While these tools can merge changes to a sizable single file in a few seconds, in big projects, document sets are large and the time to merge many documents will be substantial and irritating. Furthermore, the tools lack useful conflict resolution interfaces or APIs, which adds considerably to the effort required for practical document merging.

Our approach to three-way merging starts with the use of unique IDs so that nodes that have undergone substantial transformations can still be matched with their original version. The unique IDs also help to better distinguish move operations and to improve conflict detection. To reduce memory usage, we use a versioned tree data structure so that only one full document tree must be created in memory, along with enough tree deltas to represent any changes. The versioned tree supports a change history that allows it to merge only those nodes that have changed, while ignoring all others.

1.4 Terminology and notation for three-way tree merging

The problem of three-way tree merging can be defined as follows. Suppose t_0, t_1 and t_2 are ordered document trees where t_1 and t_2 are each derived from t_0 by separate sets of changes. t_0 is the *common base* or *common ancestor* of t_1 and t_2 in the revision tree. The problem of merging t_1 and t_2 is to find t_3 such that t_3 can be derived by a combination of the changes made to t_0 to produce t_1 and the changes made to t_0 to produce t_2. In general, the problem of merging trees involves matching the nodes between the base tree and changed trees, identifying the changes made and finding a merge of the two sets of changes. We will use this t_i notation throughout the rest of the paper.

When performing a merge of t_1 and t_2, the starting assumption is that all changes made by the respective authors are valid. But when performing the merge operations, *conflicts* can occur when two changes are made in the same location. A simple example of a conflict occurs when the two authors add different material at the same location. This is a conflict because there is no clear choice for which addition should be placed first. We use the term *false conflict* to describe independent changes by two authors at the same location that produce identical results. A serious practical solution

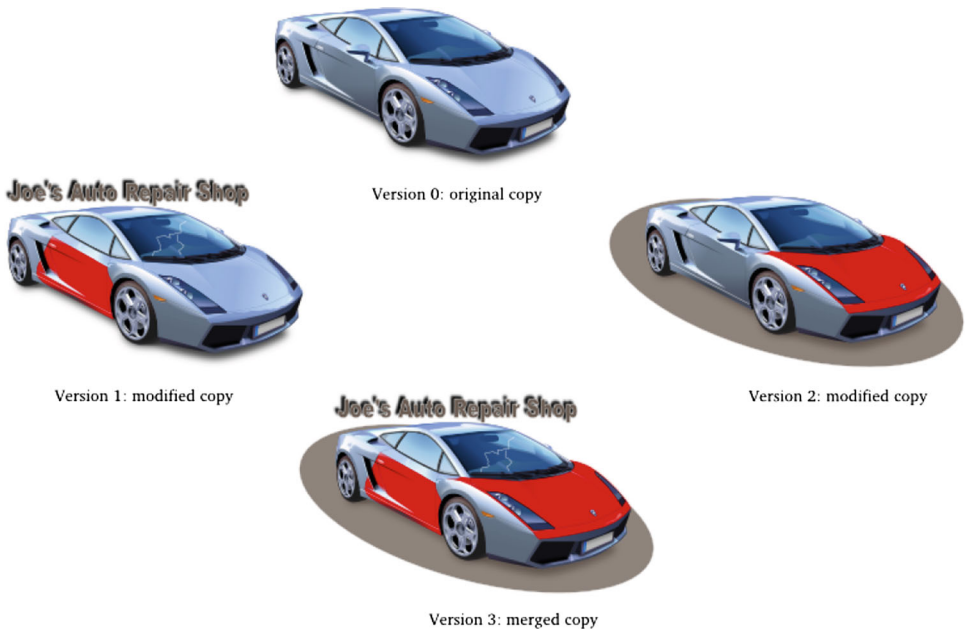


Fig. 1 A three-way merge example for SVG documents

to version control must be able identify conflicts and allow authors to resolve them. Our algorithm can identify conflicts and our prototype provides simple support for conflict resolution. However, this paper is not focused on the problem of conflicts.

Figure 1 presents a three-way merge of SVG documents. The top image (Version 0) is the original document. It has been copied and shared with two other users so that each one can add something to the original image. In Version 1, the user added the text "Joe's Auto Repair Shop", painted the side door, removed the right mirror, and drew a crack on the windshield to represent a damaged car. The second user modified his copy to create a shadow below the car and painted the hood. The third file (Version 3) is the merge result produced by the proposed algorithm.

1.5 Outline

The outline of this paper is as follows. We describe how versioned XML documents are mapped to a versioned tree data structure in Sect. 2. Section 3 describes the merge and conflict rules as well as the handling of false conflicts, node mapping and the merge algorithm. Section 4 presents an evaluation of the algorithm, while Sect. 5 discusses related work and the conclusion and future work are described in Sect. 6.

2 Versioned tree data model

In this section we describe the versioned data structure and how it is used to model XML tree and its change detection mechanism. We use versioned data structures from the Fluid project [13].

2.1 Versioned data structure

The Fluid project's [13] goal is to develop tools for Java program transformation. As part of the project, a low level versioning system was developed using Fluid's internal representation (IR). Fluid's central representations are nodes, slots, attributes and versions.

- *Nodes* are loci of identity and contain no other information.
- *Versions* are points in tree-structured discrete time. They are arranged into a tree called the *version tree* where the root is the *initial* or first version and parent versions represent older states than their children.
- *Slots* are locations that can store information including references to other slots and nodes. *Versioned slots* are specialized slots that can store different information at different versions.
- *Attributes* are names that map nodes to slots. Given a node and an attribute, we can obtain the slot value assigned to that node. This models the idea that nodes have attributes.

Nodes, attributes and slots then can be thought of as a table where the rows are the nodes, the columns are the attributes and the cells are the slots that store the values. With the addition of versions, the table becomes a three dimensional table where the third dimension is version. With versioned slots, given a node, an attribute, and a version, the value at that version is retrieved. This structure models the notion that a node's attributes have different values at different versions.

The API to the Fluid IR provides a change-and-commit model for managing versions. When a new version of a document is created, it is represented by an empty table of nodes,

attributes and slots that is the *current version*. This table is filled in with information and at some point the result is *committed* as an initial version, v_0. Once committed, v_0 never changes again in any way. Further editing changes to the "current version" are permitted, but those changes are part of a new, unnamed version. Eventually, those changes may be committed as v_1. Once more than one version has been committed into the system, it becomes possible to set the "current version" to be any such version and to start making a new version derived from it.

2.2 Change history

Changes to a node are recorded by a listener object of type *ChangeRecord*. This occurs during parsing of derived documents as described in Sect. 3.4. Using a *ChangeRecord* object, a given node n can be queried whether it has changed in v_2 relative to v_1. *ChangeRecord* objects are coarse-grained; they know that n has changed but not the nature of the change. The change could be an update to one or more attributes or an addition or deletion of one or more children. For example, if the value of an attribute a of node n is updated, n is marked as changed from the previous version, but *ChangeRecord* says nothing about a being changed other than that node n has changed from previous version. A node that has a child added or removed is marked as changed, but the child that was added or removed is not marked as changed. Using a *ChangeRecord* we can obtain a list of nodes that have been changed from a given version. This gives us the notion of change history and is used during the merge process to ignore nodes that have not changed.

2.3 XML documents as versioned trees

More complex structures such as containers and trees are formed using nodes, attributes and slots. A container is a collection of slots and can act as a linked list or as a fixed size array. A node in a versioned tree has a *children* attribute and a *parent* attribute. The parent attribute maps to a slot containing the reference of the parent node. The children attribute maps to a container containing the references of the child nodes. To represent an XML element, a node is given a *tagname* attribute, and an *attributes* attribute that points to a collection of name-value pairs which hold the name and value of an XML attribute as specified in an XML element's start tag. Notice that we are now talking about two kinds of attributes (Fluid and XML) at the same time. This is confusing, but appears unavoidable. For text nodes in the XML tree, there is one additional Fluid attribute *text* that holds the text or character data. We ignore comments and processing instructions although they could easily be represented.

Mapping the Fluid attributes to versioned slots allows us to represent trees at different versions. Figure 2 shows the modeling of two XML documents where the second is a modified copy of the first as versioned trees. For each version, the XML source is shown on the left, while the corresponding versioned tree is shown on the right. Nodes with solid circles are nodes that were created in that version. Their Fluid attribute values can be accessed by descendant versions in the version tree but not by ancestor versions. For example, nodes *a, b, c* and *d* in version 0 were created in version 0. Since version 1 is a child of version 0, it shares the nodes from version 0 which are shown with dashed circles. In version 1, only node *e* was created while node *b* was deleted. We refer to the representation of these changes as the *delta* between version 0 and version 1. Note that the tree at version 0 and version 1 is really the same tree in memory but depending on which version has been chosen as the current version, traversal of the tree will give the tree structure for that version.

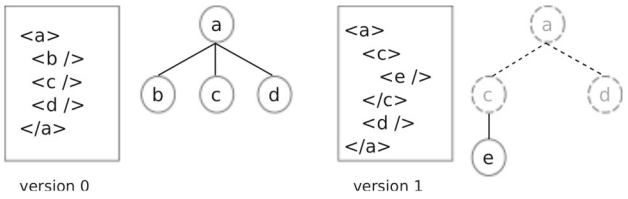


Fig. 2 Representing XML documents as versioned trees

3 Proposed merge algorithm

This section describes the merge algorithm. It starts out by describing rules for merging and conflict detection. Then it describes how elements in derived XML documents are matched to elements that already existed in the base document. Then once these preliminaries are complete, the algorithm is presented in detail.

Throughout this section, we will use t_0 to denote the base document tree, t_1 and t_2 as the two derived document trees, and t_3 as the merged document tree. v_k refers to the version for which t_k is the document tree. We also use t' to denote either t_1 or t_2, n to denote an XML element, and n' to denote an updated version of n. The assertion $n \in t_k$ says that element n is a part of tree t_k in version v_k.

3.1 Merge rules

We model the changes between versions of an XML document as operations on the base document t_0. Operations that can occur are addition, deletion, update and move. An *addition* occurs when a new element is created and added to t' which did not exist in t_0. The *deletion* of an element removes the subtree of which that element is the root. An *update* operation occurs when an element in t' sees changes in one or more XML attributes, in its tagname, or in its sequence of child elements. An element is *moved* when its parent is changed or

its position in the sequence of children of the same parent is changed. Notice that the movement of elements is marked as two operations: an update of the parent and a move of the child.

The following are the merge rules. They are written under the assumption that there are no conflicts. The rules describe the operations that will be applied to t_0 in order to produce t_3.

- *addition*: if n is added to t', then n also appears in t_3.
- *deletion*: if n is deleted in t', then n is deleted in t_3.
- *update*: if n is updated to n' in t' then n' replaces n in t_3.
- *update*: if n is updated in both t_1 and t_2 and the changes are disjoint, then all changes are made in t_3.
- *move*: if n is moved in t', the move also occurs in t_3.

3.2 Conflict detection rules

A merge algorithm must have rules for detecting conflicts. Lindholm [14] adapted the conventional conflict rules for raw text files to the context of ordered trees. We have adopted his conflict model which we describe here. Similar conflict rules have been articulated for mobile XML data by Lam et al. [15].

A *conflict* occurs when changes to the same element occur in both t_1 and t_2. For example, if n has an attribute value changed in t_1 but n is removed in t_2, there would be a conflict. The following list specifies the cases that cause conflicts. In this list, we let n_1 and n_2 to refer to respective elements in t_1 and t_2 and the elements are different. We use n to refer to a single element that is relevant in both trees.

- n_1 is added or moved to a location in t_1 and n_2 is added or moved to the same location in t_2. This is a conflict because the correct order of the two elements cannot be determined.
- n is moved in both t_1 and t_2, but its new location in those trees is not the same.
- n is updated or moved in t_1 and n is deleted in t_2.
- attribute a of n is updated with different values in both trees.
- n is deleted in t_1 and n or one of its descendants are updated in t_2.

3.3 False conflict handling

The following cases are combinations of changes that we believe should not be considered conflicts because we want to avoid overloading users with false conflicts. An alternative model would be to have conflict levels similar to the error/warning distinction in many compilers.

- attribute a of n is updated with the same content in both t_1 and t_2.
- n is deleted from both t_1 and t_2.
- n is moved to same location in both t_1 and t_2.

3.4 Node matching

In our system, node matching and change detection occur at parse time for the three documents. Our approach relies on versioning services provided by the Fluid IR. Note that we give every element in the base document a `mouid` attribute containing a unique ID string.

The base document is first read in and a versioned tree t_0 is created for it and committed as the representation of version v_0. A hash table is created mapping UIDs to elements in t_0.

Next, the parser reads in the the first modified document in order to construct a delta representing t_1. Each element is examined to see if it has a `mouid` attribute. If an element does, then that element is compared to the corresponding element in t_0. If the element has changed, then the corresponding operations are added to the delta for t_1. Otherwise, no action is taken. Note that we ignore the permutation of attributes so that elements having different permutations of attributes in their start tags will not trigger a change event. If an element read in for t_1 does not have a UID, then it is a new element and an element representing it is added to the delta for t_1. When the entire document has been processed, the delta for t_1 is committed as the representation of a new version v_1.

The system then turns to parsing the second modified XML document that will be t_2. The current version is set to be v_0 and parsing proceeds exactly as with the first document, except that change representation are added to a delta for t_2, which is committed as the representation of v_2.

Our approach requires UIDs be placed on the elements of the base document before it is shared. Stamping UIDs on the base document has a linear runtime relative to the numbers of elements. Any element in the derived documents that does not have a UID must be a new node introduced by the editor. If the editor stamps new elements with UIDs then new elements in both derived documents must not have duplicate UIDs. Since most editors do not assign UIDs to XML elements, we propose that before a document is shared, it should first be stamped with UIDs. We also assume that any editors used will preserve UIDs during editing but do not need to stamp new elements with UIDs. We have tested our approach with the Inkscape [16] and GLIPS SVG editor [17] and they both preserve our UIDs. We also propose that to better support merging, tools should stamp XML elements with UIDs when first creating a file. Our current approach is to stamp the file using our tool, and then share the file. The file can then be edited by tools that do not strip out the UIDs.

Other XML merging tools do element matching by computing hash values of each element from all three files and then matching elements that have the same hash values or

whose hash values meet a certain threshold for a closeness approximation. These approaches require creating full document trees for all three documents before elements can be matched. Furthermore, our UID-based approach is able to accurately represent radical changes to elements that make matching difficult under the hash-based model. An important restriction of the UID-based approach is that it can't merge base and derived documents that lack UIDs. Thus, our approach will not work with XML documents produced by arbitrary tools. We suggest that the performance and power advantages of the UID-based approach argue for the addition of UID support to XML editing environments and are engaged in research to show that scaleable approaches exist for doing so.

3.5 The algorithm

Once t_0, t_1 and t_2 have been constructed from the documents by the parser, the merge process can begin as described by Algorithm 1. Before the merge, t_3 is created as a new revision or a *branch* of t_2, which places t_3 in a child version of the version of t_2. t_3 and t_2 are identical initially. To merge t_1 and t_2, we look for nodes in t_1 that are marked as changed from t_0 by querying the *ChangeRecord* for t_1. For each changed element n in t_1, we get its children in t_0, t_1 and t_2. An Longest Common Subsequence (LCS) based on node IDs is computed for the sequence of children of n in t_0 and t_1 and another LCS for the sequence of children of n in t_0 and t_2. We then use a node sequence *diff3* algorithm to compute a new sequence of children for the merged node n in t_3 (Khanna et al. [18] give a formal presentation of the *diff3* algorithm). Element n's children in t_3 are then manipulated by Algorithm 2 such that n's children will have the same sequence as the merge sequence. Lastly, the attribute values for n in t_1 and t_2 are then merged, which is a much simpler process, since order of attributes is irrelevant.

Algorithm 2 describes the process of merging the children sequence from n in t_1 and t_2. Elements that are in the merge sequence, but not in the children sequence of n in t_3, are elements that have been added in t_1. To distinguish a move from an add, we check to see if the element that appears in the merge sequence already exists in t_0 and t_3. If it does not, then it is a new element in t_1 and we simply add it to t_3. We then copy the attribute values of n from t_1 because t_3 can't access attribute values of n because t_3's version is not a descendent of the version which t_1 is in. Elements that are not in the merge sequence, but are children of n in t_3 are elements that have been removed in the merge. So, these elements are removed from t_3. Once all the changed elements' children have been merged, t_3 is the tree that merges the changes from both t_1 and t_2.

```
input  : t0, t1, t2: document trees
output: t3: merged document tree

t3 ← branchOf(t2);
changeList ← getChangedNodes(t1, t0);
foreach n ∈ changeList do
    children0 ← getChildren(n, t0);
    children1 ← getChildren(n, t1);
    children2 ← getChildren(n, t2);
    lcs1 ← computeLCS(children0, children1);
    lcs2 ← computeLCS(children0, children2);
    mergeSeq ← computeMergeSeq(lcs1, lcs2);
    mergeAttributes(n, t0, t1, t3);
    mergeChildren(mergeSeq, n, t0, t1, t2, t3);
end
return t3 ;
```

Algorithm 1: Three-way merge

```
input  : mergeSeq, n, t0, t1, t2
output: n ∈ t3 with its children sequence matching mergeSeq

removeChildren(n, t3);
foreach c ∈ mergeSeq do
    if c ∉ t3 ∧ ∉ t0 then
        c ← copyNodeContent(c, t1, t3);
    else
        parent ← getParent(c, t3);
        removeChild(parent, c, t3);
    end
    addChild(n, c, t3);
end
```

Algorithm 2: Merge children

The runtime and space complexity of the algorithm are quadratic. In the worst case, in which the tree is one level deep and there are N nodes, then the runtime is $O(N-1)^2$. Space complexity is also $O(N-1)^2$ in the worst case since we use a quadratic LCS algorithm. The versioned tree data structure reduces memory usage when there are elements in the modified documents that are also in the base document. When there are no nodes common between all three documents, our approach will build three complete trees, but this seems to be a rare and degenerate case.

The tree data structures throughout the merge algorithm look like those in Fig. 3. Throughout the entire process from parsing all the way to the end of the merge, there is just one full tree and some delta nodes that are new in t_1 and t_2. There is no need to create any new node for t_3 as all of its nodes are from t_1 and t_2. Note that node f in t_3 has a dashed circle but a solid label and a solid edge linking it to the tree. This denotes that the Fluid IR node for the element itself is shared between t_3 and t_1, but since t_3 is not a child version of t_1, it can not access the Fluid attribute values of f in t_1. Hence, the Fluid attribute values must be copied into the delta for t_3 where they appear as newly added information.

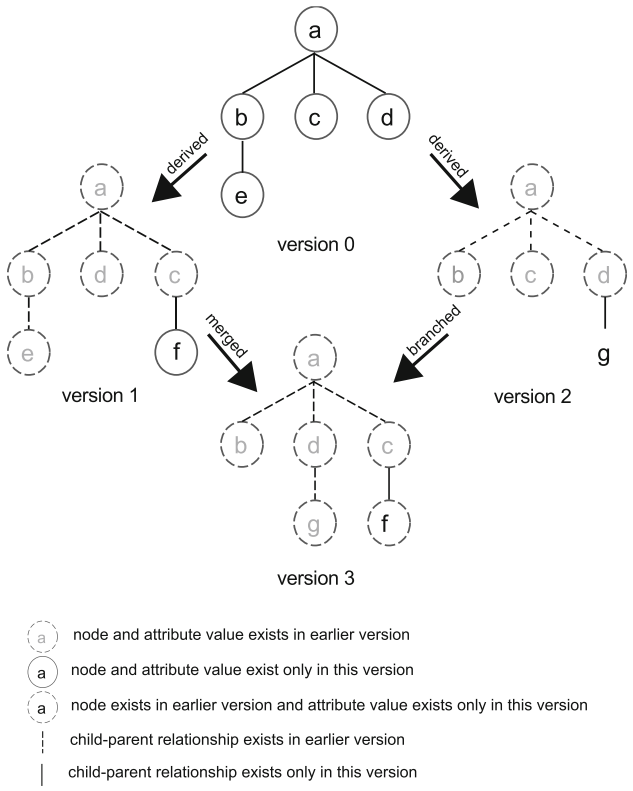


Fig. 3 Three-way merged of versioned tree

3.6 Implementation

The proposed three-way XML tool is implemented in Java and its conflict resolution interface is implemented using Java Swing. The tree and node conflict visualization make use of the Netbeans `Node`, `Explorer` and `Action` API taken from the Netbeans Platform [19]. The document parser was implemented using the Simple API for XML (SAX) API.

The tool is command-line based but it provides a graphical interface for resolving conflicts as shown in Fig. 4. Existing XML three-way merge tools, such as 3dm [12], generate only log files with cryptic messages, which makes it difficult for the user to locate conflicts in large and complex XML files. When the merge tool detects one or more conflicts, it displays the conflict resolution interface and expands those elements that are in conflict. As the user places the cursor over a conflicting element in the interface, a caption is displayed that specifies the type of conflict: attribute update conflict, insertion/move location conflict, or deletion conflict. For attribute conflicts, the user can select a node and edit the conflicting attributes using the attribute editor. The attribute editor displays only the conflicting attributes of the selected elements. The interface allows the user to move and delete elements and to update an element's name in order to resolve conflicts.

Figure 4 displays the conflicts that resulted from merging two Inkscape [16] SVG documents derived from the

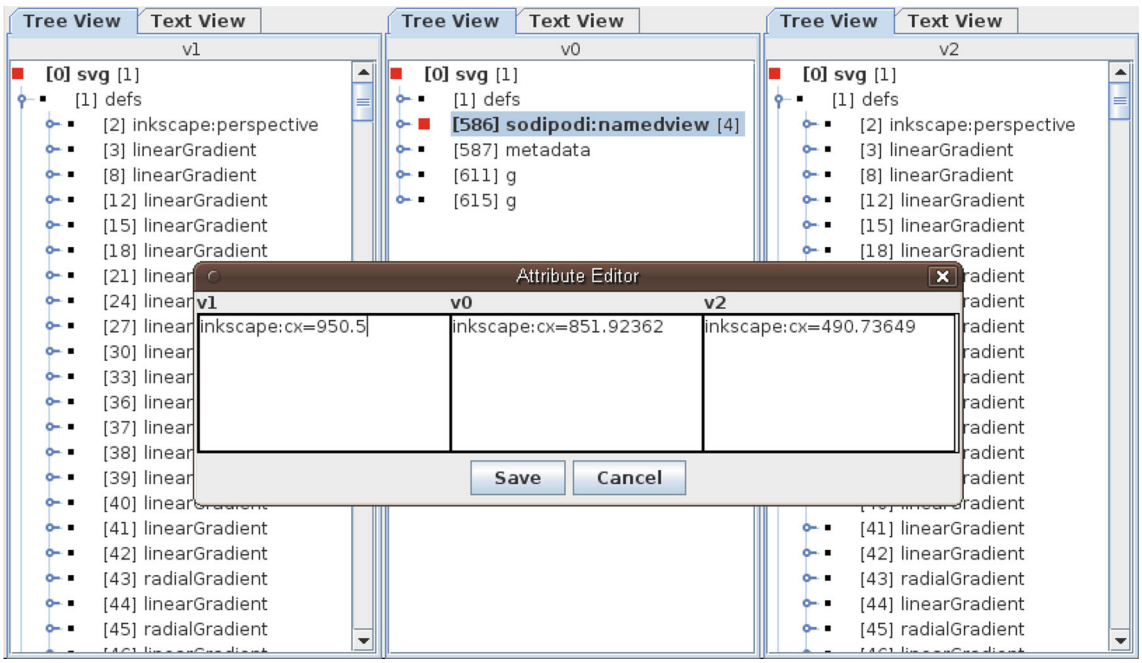


Fig. 4 Graphical conflict resolution interface. Conflicting elements are marked with a *large square* in front of the element name. The number displayed after a conflicting element helps the user identify the conflicting elements in the other trees. The number in front of the element is the node ID and allows users to match elements in different trees especially when the element name is not meaningful and there are many elements with that name

same document. The *svg* element conflict is due to the filename attribute because Inkscape stores the filenames in the SVG document itself and these two files were saved with different names. Inkscape also saves the window's coordinates in the SVG document, which results in merge conflict in *sodipodi:namedview* element. The editor displays an attribute editor for the conflicting node *sodipodi:namedview*. The conflict here is the attribute *inkscape:cx* which represents the x coordinate of the window that was saved. Not shown in the figure is the conflict between v_1 and v_2 due to adding two gradient nodes to the same location. Hence, the interface expands both trees in v_1 and v_2 to show the conflicts but not v_0 since v_0 does not have any of these nodes.

The interface also provides a text view of each of the tree documents. The user may choose to edit the text rather than using the tree view editor. Unlike some source code merging software, the current implementation does not provide color markup to highlight conflicts.

4 Performance evaluation

In this section we present an experiment to evaluate the performance of the our merge tool in terms of speed, memory usage and scalability relative to existing research and commercial three-way XML merge tools. For three-way merge tools, we are only aware of *3dm* [12] and *deltaxml* [11]. For this experiment, we used the trial version of *deltaxml*, which is limited to documents with ten thousand elements. We also include *xcc* [20] in this experiment, which is a two-way diff tool and can apply its delta to XML files that the delta was not originally created for. This feature allows *xcc* to provide a limited form of three-way merging and in many cases xcc produced incorrect merges. Still, the performance of xcc should act as a good point of comparison since it does less work than a three-way merge tool.

The questions the experiment tries to answers are: (a) whether the use of the versioned tree structure reduces overall memory usage, (b) whether the use of the versioned tree structure speeds up merging, (c) how the algorithm scales with increases in both the number of elements and the number of changes to the document.

4.1 Test data and hardware configuration

To test the performance of our proposed algorithm, we created a set of five automatically generated XML base documents and created derived versions based on random editing changes.

Five base documents were generated at sizes of 2,000, 4,000, 6,000, 8,000, and 10,000 elements. Every element had both a unique name and a unique identifier attribute, plus four attributes `a0`–`a3` whose values were random strings of digits. The trees were of depth 6 or 7 and had a fixed branching factor of 4, except that some nodes had fewer children because of the round-number sizes of the trees. These trees are not particularly realistic, but we lacked a standard for realism and decided, after some consideration, that these synthetic trees were a sufficient basis for performance testing of algorithms.

For each base document, we created sets of editing changes at six different sizes: 0, 20, 40, 60, 80, and 100. We divided these sets into equal-sized halves and applied them to the base documents to create two derived versions for each base document. The change size of zero was a degenerate case where the so-called derived documents were identical to the base document. Each set of modifications was divided into updates, deletions, additions, and moves where deletions, insertions, and moves were allotted 20 % of the modifications and the remaining 40 % of modifications were updates to attribute values. The nodes and attributes selected to be modified were chosen randomly using a uniform distribution among nodes and attributes, but we also ensured, by a combination of automatic and manual means, that there were no conflicts between the change sets for any tool. The total numbers of files created was 55 including the base versions.

The experiment was conducted on a Lenovo Thinkpad X61 with 2.2 Ghz Core 2 Duo processor and 4 GB of RAM, running Ubuntu 10.04 Beta 1 AMD 64 with a Solid State Disk (approximately 100 MB/s read). The code was built and run on OpenJDK 1.6 as shipped with Ubuntu 10.04. All the tools ran with default settings (`-Xmx` variable was not set).

We use `System.currentTimeMillis()` as a way to determine the amount of time a segment of code executes and `totalMemory` and `getFreeMemory` methods of the Runtime class to determine the amount of memory currently used in the Java Virtual machine. We modified *3dm*'s code to print execution time and memory usage. For *deltaxml*, we used the `PipelinedSynchronizer` API of the *deltaxml* package to do the merge. Since we do not have the source code, we can only measure the entire execution from parsing to saving the merge result. This prevents us from determine how fast *deltaxml* parses XML files and writes merged result to disk compared to *3dm*, and *xcc*.

Note that we use *total time* to mean the total execution time including parsing, merging, and writing the result to disk. When we say *actual merge time* we refer only to the execution of the merge algorithm excluding the parse time and the time spent saving the merge output. For this discussion, our implementation is simply called *molhado*.

4.2 Results

First, we compare the execution time and memory usage of molhado, xcc, 3dm and deltaxml as the number of changes increases. The number of elements for this test is fixed at

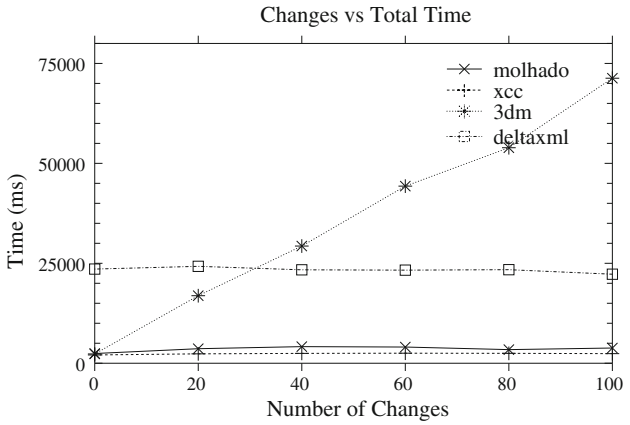


Fig. 5 Total execution time as changes increase

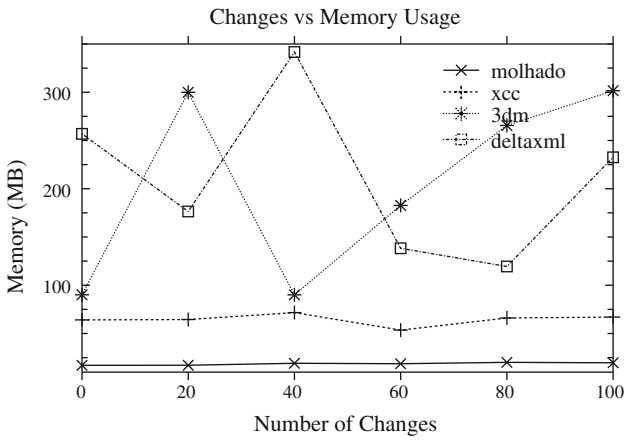


Fig. 6 Memory usage as changes increase

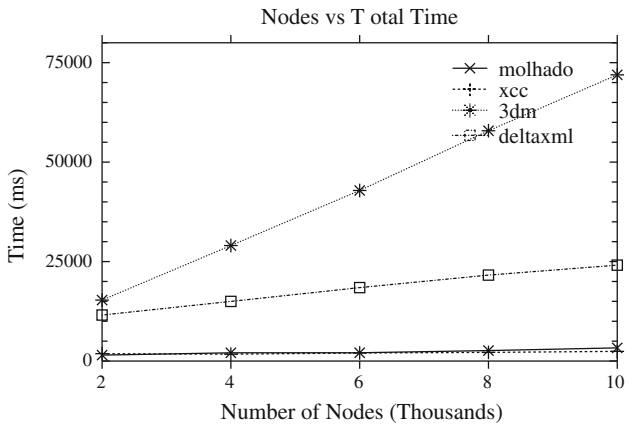


Fig. 7 Total execution time as elements increase

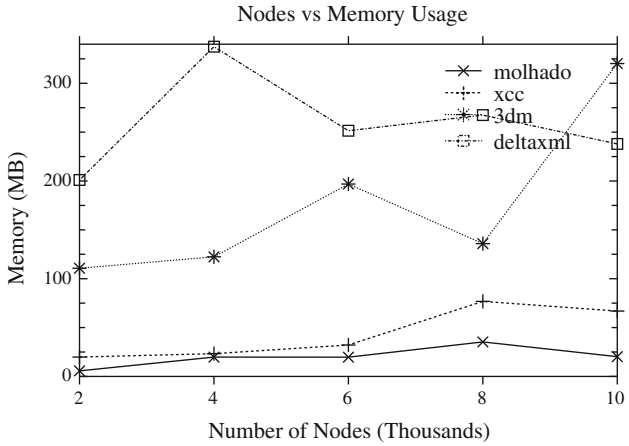


Fig. 8 Memory usage as elements increase

10,000 while the changes are increased by increments of 20. Figure 5 compares the different tools' total execution time with increasing changes. The charts shows that molhado and xcc have the lowest total execution time. 3dm's execution time increases rapidly and linearly with an increase in changes while all other tools' execution times increase by a very small amount. As shown in Fig. 6, molhado and xcc show an almost constant level of memory usage relative to the number of changes. 3dm and deltaxml show high variability of memory usage with results suggesting that memory usage is increasing for 3dm at higher levels. Molhado uses the least memory among all the tools which appears to confirm that the versioned tree data structure reduces memory requirements.

To test whether document size affected total time and memory usage, we fixed the number of changes at 100 while the number of elements increased from 2,000 to 10,000 in steps of 2,000. The results are plotted in Figs. 7 and 8. In this test, molhado and xcc show nearly identical and low total processing times. They also have low memory usage, though there is suggestion that the memory usage of xcc is growing somewhat faster than that of molhado. In constrast, both deltaxml and 3dm appear significantly affected by the number of elements. In total time, deltaxml appears to have a substantial startup cost in total time, with a modest linear increase as the number of elements increases. 3dm's total time shows a dramatic linear increase with the number of elements. 3dm merges using all elements regardless of whether they have changed, thus increasing merge time linearly. Both 3dm and deltaxml have high memory usage relative to molhado and xcc.

Figures 9 and 10 plot just the execution of the merge algorithm excluding the time to parse and writing of merge result to disk. Deltaxml was not included because we do not have its source code and were unable to exclude its parse and writing time. In this comparison, 3dm is affected by both increased in changes and the number of elements in linear manner. Although it is hard to see how the number of changes affect molhado in Fig. 9, the execution growth rate of molhado against changes is shown in detail in Fig. 11 which shows quadratic scaling in the number of changes. This is probably due to the LCS algorithm we used which has a worst case

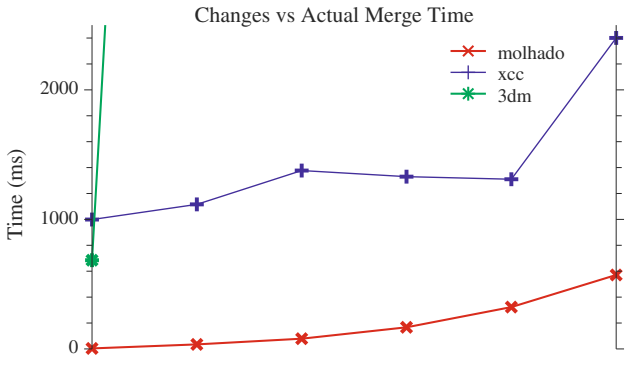


Fig. 9 Merge time as changes increase

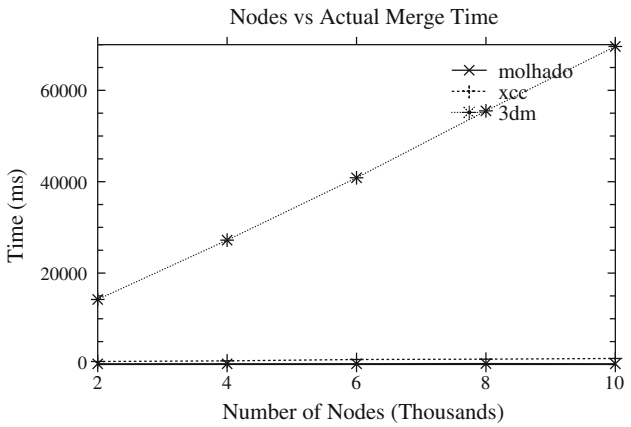


Fig. 10 Merge time as elements increase

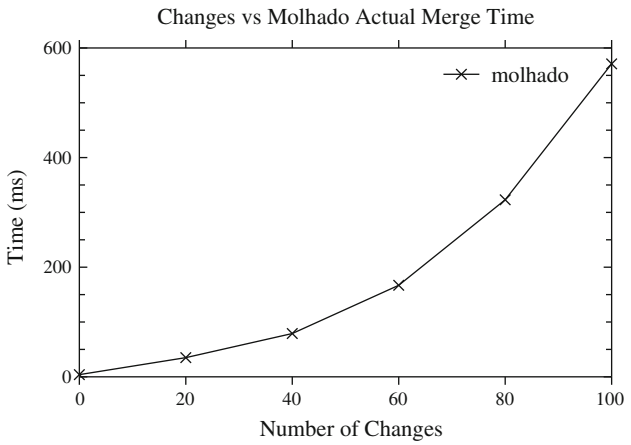


Fig. 11 Merge time as changes increase

complexity of $O(n^2)$. Figure 12 shows the merge time vs the increase in the number of elements. There is less than 100 ms increase in total time as the number of elements increases from 2,000 to 10,000. This small increase is probably for the traversal of the elements by the *ChangeRecord* data structure, as it takes more time to traverse the larger tree looking for changed elements.

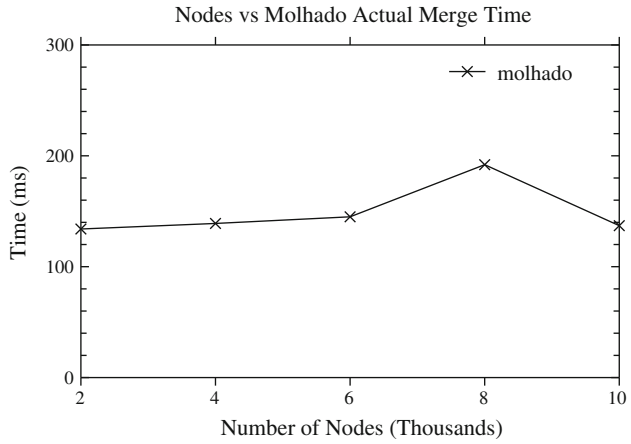


Fig. 12 Merge time as elements increase

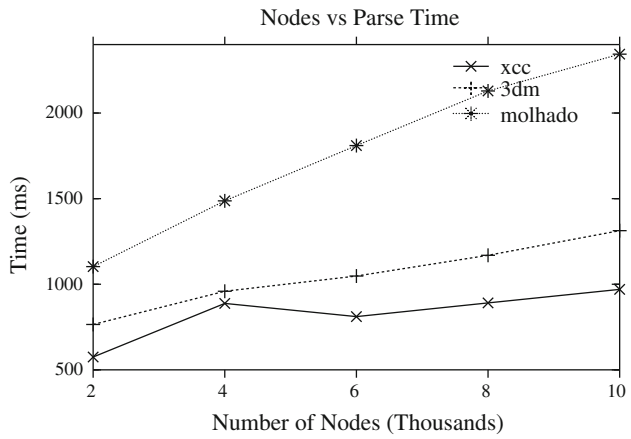


Fig. 13 Parse time as elements increase

Figure 13 shows the parse time for each tool against the number of nodes. The graph shows that molhado has the slowest parser. This is probably because building the versioned tree data structure is more expensive than building a simple DOM tree. Also, during the parsing process, elements in derived trees are marked as changed or not. This requires code that checks whether elements in derived trees are present in t_0 and whether they have changed. Notice that this matching task happens during the "parsing" process in molhado, but is part of the merging step in the other tools. As Fig. 7 shows, the large parse time of molhado is dominated by its fast merge time, making it faster than the other tools in total execution time.

On a side note, under formal XML semantics, the permutations of attributes in a start tag should not have significance. Some XML pretty print tools rearrange the attributes as documents are saved. 3dm reports rearrangement of attributes as a conflict, stating that the element has been updated, even though the two versions are semantically equivalent. 3dm's performance suffers drastically with permutation of attributes. For example, we performed a simple permutation

of the attributes of all elements in the 2,000-element file and this resulted in a 16 min (960 s) merge time for 3dm while the same files without the change in attribute order could be merged in 1.3 s. In contrast, molhado performed the same tasks in 1.15 and 1.05 s.

Overall, this experiment shows that the molhado merge algorithm is faster than that of the available tools and that the versioned tree data structure reduces memory usage. We also note that while xcc approaches molhado's performance in both time and memory, it is a more limited tool that does not perform a true three-way merge.

4.3 Summary

Overall, molhado has superior memory and runtime performance to both the 3dm and deltaxml merging algorithms. Molhado's performance is quite similar to that of the xcc patching algorithm, but patching is fundamentally a simpler task. Some more specific observations are:

- At the range of document sizes and number of changes tested, all of the algorithms appear to show linear scaling, with the differences between them being in the coefficient of scaling.
- Molhado's memory utilization is dramatically lower. This is almost certainly due to the efficient versioned tree data structure that we use, since additional storage is only created for changed nodes in the document tree.
- Close examination of merge time alone shows that Molhado's performance is quadratic in the number of changes. This seems reasonable given the quadratic behavior of the LCS algorithm. We suspect that 3dm also has a quadratic scaling factor but that this is dominated by the linear effects of building multiple trees and matching nodes at the scale of our simulated documents.

5 Related work

Mens [21] gives a survey on the state of software merging. The algorithm for *diff2* line based content is described in detail by Myers [22]. Other researchers [12,23–26] describe two-way structural differencing algorithms that make use of two documents without a base document. Their node matching techniques are based on node values and various types of hash values. Tools for three-way differencing and merging of XML documents also exist [11,12].

The GNU *diff* utility computes the differences between any two text files using the LCS algorithm [22].

Chawathe et al. [24] describe an algorithm that does not assume the use of unique IDs, but makes use of them if they are available. Cobena et al. [25] implemented a two-way XML document differencing tool that uses an ID attribute for node matching. It also make uses of a node signature, which is a hash value for the node and its content. Wang et al. [27] proposed X-Diff which uses standard tree-to-tree correction techniques. X-Diff treats XML documents as unordered trees, which is not suitable for most document applications. diffX, which was described by Al-Ekram et al. [23], matches nodes using node types and node name. Two nodes are equal if both their types and labels are equal. Lanham et al. [26] described an algorithm called vdiff. It makes use of the node's unique ID and hash values for matching. Lindholm et al. [28] described an XML differencing algorithm that works on a sequence of token encoding XML documents rather than document trees. It computes a match list from the input token sequences using a sequence alignment algorithm and hash values.

DeltaXML [11], a commercial tool, supports three-way merge of XML documents. Node matching is done using node ID and longest common subsequence alignment at each level of the input trees. By default, the order of children within a node is important, but it could be ignored. DeltaXML supports the addition, deletion and update operations.

Lindholm et al. [12] presented a three-way merge algorithm for XML documents that performs tree to tree mapping. An XML document is encoded as a set of content and parent–child–successor (PCS) relations. A set of changes are consistent if the changes in the set are unambiguous. The set of changes are combined into a "raw" merge and then inconsistencies are removed by iterating over the "raw" merge to create the change set for the merge. It supports the addition, deletion, update, and move operations. Nodes are required to match to at most one node. The algorithm is attractive in its simplicity, but it lacks the ability to deal with multiple moves in either modified tree. When there are multiple moves that are within the same parent, the change list will be inconsistent. The algorithm simply ignores the position. In our approach, the positions of the merged children are computed by using *diff3* which places the children in the correct order, and if it cannot determine the position of children, it is marked as a conflict.

Rönnau et al. [20] describes a differencing and merging algorithm for XML documents using context fingerprints, a sequence of hash values of all nodes within certain distance from the edit operation, to help resolve the location of operations during merging. The algorithm supports the add, delete and update operations. The hash values are computed using the node element name and its sorted attributes. This means that a node which has its element name changed will have a different hash value, hence appearing to be a different node. Using node identity, our approach knows the node is the same even if its element name and its attribute values change. Instead of performing a delete and insert operation, it does an update. In Rönnau's approach, finding the node for the merge operation is heuristic where in our approach, we know

on what node the operation must occur because of node identity and *diff3*. The complexity of Rönnau's approach results from trying to support merging of documents the difference has not been computed, thus allowing it to perform a limited three-way merge.

Westfechtel [29] and Schwägerl et al. [30] proposed and implemented a three-way merging tool for models in the Eclipse Modeling Framework (EMF). Although, EMF models are encoded as XML, the merge tool uses semantics specific to EMF models in order to ensure that merge results are valid. This contrasts with our algorithm which uses less-specific, generic XML semantics.

Abdessalem et al. [31,32] have proposed a probabilistic XML merging tool. Changes are associated with probabilities based on the author's credentials. This approach is proposed to support a kind of adaptive shared document for which readers can assign measures of trustworthiness to different authors, so that each reader sees a document composed of changes made by authors that the reader most trusts. Our algorithm has no probabilistic aspects and all authors' changes are weighted equally.

Vion-Dury [33,34] proposed a generic calculus of editing deltas but no tool has been implemented to verify the concept.

6 Conclusion and future work

We have described an approach to three-way XML document merging using a versioned tree data structure, change detection and node identity. We showed that it is faster and uses less memory than other three-way XML merge tools. Unlike other approaches, which create all elements for three or four trees, our approach creates one full tree t_0, delta nodes for t_1 and t_2 and no nodes for t_3. With change detection, the algorithm only merges changed nodes, reducing the number of LCS computations. The evaluation experiment shows that the algorithm's performance is largely unaffected by the number of nodes. The algorithm's runtime is affected most by the number of changes and this appears to be mainly due to the LCS algorithm.

Future work on this algorithm and its implementation should include adjusting the algorithm to gain further improvement in both speed and memory usage. The current implementation uses the simple standard LCS algorithm. Replacing it with the LCS algorithm described by Myers [22], which has a linear space complexity, could further reduce memory usage. Merge performance can further be improved by reducing the numbers of nodes to be merged. Choosing the derived tree (t_1 or t_2) that has the most changes and deriving t_3 from it could well reduce run time. Since the algorithm only visits nodes changed between t_0 and the one derived tree that is not the parent of t_3, there would be less nodes to visit. This is only an advantage when the sizes of the change sets of the two trees are different by a large degree. Refining change detection so that it can distinguish the kind of change (changes to attributes vs. changes to the child list) might reduce the number of elements for which it is necessary to compute the children merge sequence. Other things to consider include the use of XML schema to improve correctness of merging, incorporating the use of hash values for elements when there exist no UIDs in all three documents, and allowing the user to specify whether the order of attributes should be preserved. Finally, the evaluation experiment could be expanded to a larger set of simulated documents so that any doubt about the relevance of the performance results could be eliminated.

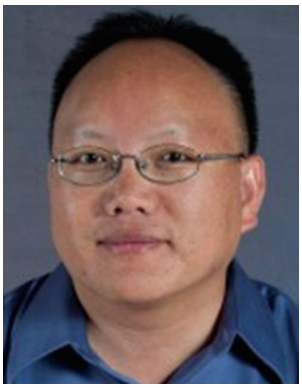
In the area of applications, the XML three-way merge algorithm is now part of a set of XML versioning tools on which we have based other software. We have constructed version control tools for software product line engineering [35] that use the same versioning infrastructure. In other work, we have developed a new add-on structure for XML documents that places a version history inside each document file and we call the resulting file format *version aware documents* [36]. The add-on structure uses XML namespaces to include a preamble with the version history and a set of UIDs on the document nodes. We are now modifying LibreOffice [37] to accept our format in order to demonstrate that version aware documents are practical and that realistic systems will not require radical changes to support them [38].

Acknowledgments This research has been supported by an HP Labs Innovation Research Grant.

References

1. International Digital Publishing Forum (2010) http://idpf.org/epub/. Accessed 15 June 2010
2. MOF 2 XMI Mapping (XMI) (2013) http://www.omg.org/spec/XMI/. Accessed 2 May 2013
3. Collard ML, Maletic JI, Marcus A (2002) Supporting document and data views of source code. In: Proceedings of the 2002 ACM symposium on document engineering, DocEng '02. ACM, New York, pp 34–41.
4. Rochkind M (1975) The source code control system. IEEE Trans Softw Eng 1(4):364–370
5. Tichy WF (1985) RCS–a system for version control. Softw Pract Exp 15(7):637–654
6. Morse T (1996) CVS. Linux J 21es:3.
7. Subversion (2013) http://subversion.tigris.org/. Accessed 2 May 2013
8. Mercurial SCM (2013) http://mercurial.selenic.com/. Accessed 2 May 2013
9. Git (2013) http://git-scm.com/. Accessed 2 May 2013
10. GNU diff3 (2013) http://www.gnu.org/software/diffutils/. Accessed 15 June 2010
11. Fontaine RL (2002) Merging XML files: a new approach providing intelligent merge of XML data sets. In: Proceedings of XML Europe 2002

12. Lindholm T (2004) A three-way merge for XML documents. Proceedings of the 4th ACM symposium on document engineering. ACM Press, New York, pp 1–10
13. Boyland J, Greenhouse A, Scherlis WL (2005) The fluid IR: an internal representation for a software engineering environment. http://www.fluid.cs.cmu.edu. Accessed 15 June 2010
14. Lindholm T (2001) A 3-way merging algorithm for synchronizing ordered trees–the 3dm merging and differencing tool for xml. Master's thesis. University of Helsinki, Helsinki
15. Lam F, Lam N, Wong R (2002) Efficient synchronization for mobile xml data. In: Proceedings of the 11th international conference on information and knowledge management, CIKM '02. ACM, New York, pp 153–160
16. Inkscape SVG editor (2013) http://www.inkscape.org/. Accessed 15 June 2010
17. Glips graffiti editor (2013) http://glipssvgeditor.sourceforge.net/. Accessed 15 June 2010
18. Khanna S, Kunal K, Pierce BC (2007) A formal investigation of diff3. In: Arvind V, Prasad S (eds) Foundations of software technology and theoretical computer science (FSTTCS)
19. Netbeans platform (2013) http://platform.netbeans.org. Accessed 15 June 2010
20. Rönnau S, Pauli C, Borghoff UM (2008) Merging changes in xml documents using reliable context fingerprints. Proceeding of the 8th ACM symposium on document engineering, DocEng '08. ACM, New York, pp 52–61
21. Mens T (2002) A state-of-the-art survey on software merging. IEEE Trans Softw Eng 28(5):449–462
22. Myers EW (1986) An O(ND) difference algorithm and its variations. Algorithmica 1:251–266
23. Al-Ekram R, Adma A, Baysal O (2005) diffX: an algorithm to detect changes in multi-version XML documents. In: Cordy JR, Kark AW, Stewart DA (eds) CASCON. IBM, UK, pp 1–11
24. Chawathe SS, Rajaraman A, Garcia-Molina H, Widom J (1996) Change detection in hierarchically structured information. Proceedings of the 1996 ACM SIGMOD international conference on management of data, SIGMOD '96. ACM, New York, pp 493–504
25. Cobena G, Abiteboul S, Marian A (2002) Detecting changes in XML documents. In: Proceedings of the 18th international conference on data engineering, pp 41–52.
26. Lanham M, Kang A, Hammer J, Helal A, Wilson J (2002) Format-independent change detection and propogation in support of mobile computing. In: Proceedings of the XVII symposium on databases (SBBD 2002), pp 27–41.
27. Wang Y, DeWitt DJ, Cai J (2003) X-diff: an effective change detection algorithm for XML documents. In: Proceedings of the 19th international conference on data engineering. Bangalore, India, pp 519–530
28. Lindholm T, Kangasharju J, Tarkoma S (2006) Fast and simple XML tree differencing by sequence alignment. Proceedings of the 2006 ACM symposium on document engineering, DocEng '06. ACM, New York, pp 75–84
29. Westfechtel B (2010) A formal approach to three-way merging of emf models. In: Proceedings of the 1st international workshop on model comparison in practice, IWMCP '10. ACM, New York, pp 31–41
30. Schwägerl F, Uhrig S, Westfechtel B (2013) Model-based tool support for consistent three-way merging of emf models. In: Proceedings of the workshop on ACadeMics tooling with eclipse, ACME '13. ACM, New York, pp 2:1–2:10
31. Abdessalem T, Ba ML, Senellart P (2011) A probabilistic xml merging tool. In: Proceedings of the 14th international conference on extending database technology, EDBT/ICDT '11. ACM, New York, pp 538–541
32. Ba ML, Abdessalem T, Senellart P (2013) Uncertain version control in open collaborative editing of tree-structured documents. Proceedings of the 2013 ACM symposium on document engineering, DocEng '13. ACM, New York, pp 27–36
33. Vion-Dury J-Y (2010) Diffing, patching and merging xml documents: toward a generic calculus of editing deltas. Proceedings of the 10th ACM symposium on document engineering, DocEng '10. ACM, New York, pp 191–194
34. Vion-Dury J-Y (2011) A generic calculus of xml editing deltas. Proceedings of the 11th ACM symposium on document engineering, DocEng '11. ACM, New York, pp 113–120
35. Thao C (2012) A configuration management system for software product line. PhD thesis. University of Wisconsin-Milwaukee, Milwaukee
36. Thao C, Munson EV (2011) Version-aware XML documents. Proceedings of the 11th ACM symposium on document engineering, DocEng '11. ACM, New York, pp 97–100
37. LibreOffice (2013) http://libreoffice.org/. Accessed 2 May 2013
38. Pandey M, Munson EV (2013) Version aware libreoffice documents. Proceedings of the 2013 ACM symposium on document engineering, DocEng '13. ACM, New York, pp 57–60



Cheng Thao is Assistant Professor in the Computer Science Department at the University of Wisconsin-Whitewater. He received his B.S. in Human Biology from the University of Wisconsin-Green Bay, and his M.S. and Ph.D in Computer Science from the University of Wisconsin-Milwaukee. His research interests are in software configuration management, software product line, and collaborative editing.

Ethan V. Munson is Professor and Co-Chair of Computer Science in the Department of Electrical Engineering and Computer Science at the University of Wisconsin-Milwaukee, where he is also the Director of the Multimedia Software Laboratory and of the Center for Advanced Computational Imaging. He received the M.S. (1989) and Ph.D. (1994) in Computer Science from the University of California, Berkeley. His research focuses on configuration management for software and documents and on human-computer interaction. Dr. Munson is a recipient of a National Science Foundation CAREER award, as well as four NSF educational grants, and a variety of industrial funding. He is a Senior Member of ACM and a member of the Brazilian Computer Society (SBC). He was Chair of ACM SIGWEB from 2006 to 2011.