A Three-way Merge for XML Documents

Tancred Lindholm
Helsinki Institute for Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
tancred.lindholm@hiit.fi

ABSTRACT

Three-way merging is a technique that may be employed for reintegrating changes to a document in cases where multiple independently modified copies have been made. While tools for three-way merge of ASCII text files exist in the form of the ubiquitous diff and patch tools, these are of limited applicability to XML documents.

We present a method for three-way merging of XML which is targeted at merging XML formats that model human-authored documents as ordered trees (e.g. rich text formats, structured text, drawings, etc.). To this end, we investigate a number of use cases on XML merging (collaborative editing, propagating changes across document variants), from which we derive a set of high-level merge rules. Our merge is based on these rules.

We propose that our merge is easy to both understand and implement, yet sufficiently expressive to handle several important cases of merging on document structure that are beyond the capabilities of traditional text-based tools. In order to justify these claims, we applied our merging method to the merging tasks contained in the use cases. The overall performance of the merge was found to be satisfactory.

The key contributions of this work are: a set of merge rules derived from use cases on XML merging, a compact and versatile XML merge in accordance with these rules, and a classification of conflicts in the context of that merge.

Categories and Subject Descriptors

I.7.1 [**Document and Text Editing**]: Document Management; F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures; H.5.3 [**Group and Organization Interfaces**]: Collaborative computing

General Terms

Algorithms, Design, Performance

DocEng'04, October 28–30, 2004, Milwaukee, Wisconsin, USA.

Keywords

XML, three-way merge, structured text, collaborative editing, conflict

1. INTRODUCTION

One of the fundamental scenarios in computer-supported collaborative work is the parallel modification of copies of a document, and the subsequent reintegration of the copies into a single document containing the modifications [1]. We investigate the variant where two replicas of a *base* document have been made and where the replicas have been independently edited. We will refer to the replicas as the *modified* documents.

In this paper we apply reintegration based on the technique of *three-way merging* to the scenario. The technique requires the base document to be present during reintegration. The base document is required, since in three-way merging we *detect* the edits between the base and modified versions as an integral part of the merge, rather than use a trace of the actual edit operations.

An important aspect of a three-way merging algorithm is the data structure it is designed for, e.g. sets of tuples (for relational databases), ordered lists (for text files), trees and graphs (for structured data), or application-specific structures. For ASCII text there exists a ubiquitous three-way merge in the form of the diff, patch, and diff3 tools [10, 6]. Their applicability to structured data is, however, limited due to their list-based data model. A three-way merge specifically designed for structured data is thus called for.

The tree is one of the most frequently used data structures in computer science, and a widely used format for encoding trees has emerged with the advent of the Extensible Markup Language [13] (XML) in recent years. The tree as an abstract data structure, and its encoding as XML, thus seem to be the logical starting point for a generic three-way merge for structured data.

Trees may be manipulated in more complex ways than ordered lists of text lines. Consequently, a tree merge will need to handle more complex combinations of changes than a line-based tool. The correct output will depend on the semantics of the data in several of these cases (e.g. ordered and unordered trees), and thus we must conclude that there is no single "right" definition of a generic XML merge.

We do nevertheless feel that the class of XML formats that represent human-authored documents that are naturally modeled as *ordered trees* (e.g. rich text formats, drawing formats) is a sufficiently large class with enough common traits to allow for a generalized merge. In this paper, we

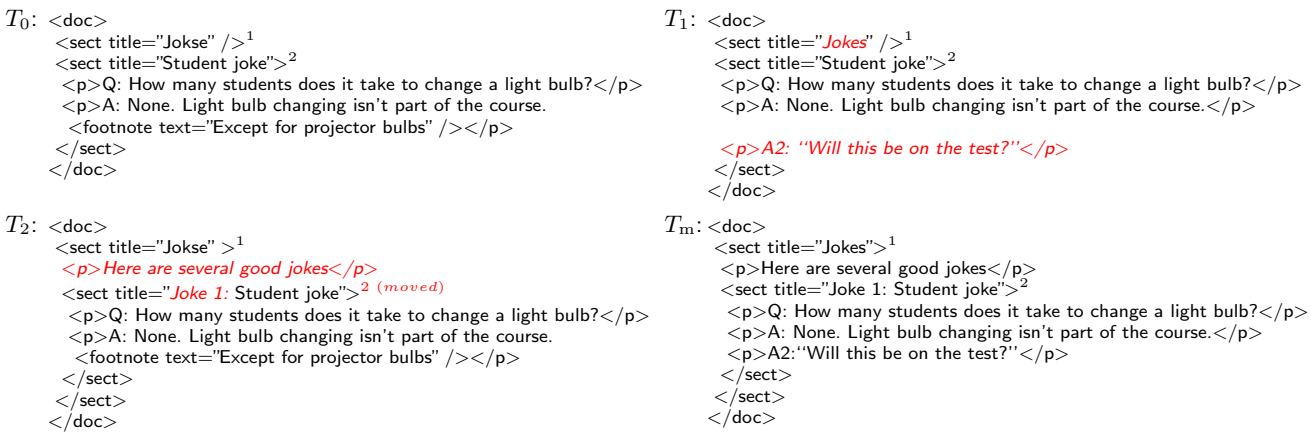

Figure 1: Merge of a structured document. Changes are marked *like this*, and superscripts are used to track the <sect> nodes.

present a merge for this class of so-called *document-oriented* XML. The merge is synthesized from a number of "real-world" use cases, and should therefore be practically useful.

The approach may be criticized for ignoring the semantics of the individual formats. To counter, we point out that an understanding of the full semantics of the data is frequently not required to perform a successful merge. For instance, consider three-way merging of program source code using the diff3 tool: the semantic structure of the source code may be very complex, yet we often obtain valid results from diff3, which considers the structure of its input to be an ordered list of text lines.

An important goal of the merge was to make it behave in an intuitive manner, in order to make it attractive as a tool for everyday use. To this end, we use an easily understood model for detecting changes, and deliberately make the merge report conflicts in ambiguous situations.

The paper is organized as follows: we introduce three-way merging of XML, along with two examples and notation in section 2. The use cases and the rules on which we base our merge are presented in section 3. Having introduced the concepts of tree matching (section 4) and changes (section 5), we state the definition of the merge in section 6. We evaluate an implementation of the merge in section 7, and review related work in section 8. Section 9 concludes.

2. THREE-WAY MERGING OF XML

Assume that we have two initially identical replicas of an object T_0, denoted T_1 and T_2, which have subsequently been independently edited. Let the *edit scripts* (see e.g. [3]) $\mathcal{E}_1$ and $\mathcal{E}_2$ be the ordered sequences of edit operations that created T_1 and T_2 from T_0. We wish to obtain a version of the object that integrates the modifications made to both replicas, i.e. we want to *merge* the changes made to the replicas into a unified version.

We informally define the result of the three-way merge of T_0, T_1, and T_2 to be the object which is obtained by applying $\mathcal{E}_1$ and $\mathcal{E}_2$ to T_0. We denote the merged object T_m.

We may identify two phases of the three-way merge process: *edit detection* and *reconciliation*. The edit detection phase is necessary since we do not assume any knowledge of $\mathcal{E}_1$ and $\mathcal{E}_2$. During this phase, a set of edits that approximate $\mathcal{E}_1$ and $\mathcal{E}_2$ is derived. The set of detected edits is then applied to T_0 during the reconciliation phase. If edit detection is well implemented, the result of the reconciliation will be the same as if it was made using $\mathcal{E}_1$ and $\mathcal{E}_2$.

As we operate on document-oriented XML we assume that T_0, T_1, T_2, and T_m are *trees* and, furthermore, that these are *ordered*. T_0 is used to denote the base tree, T_1 and T_2 are used for the modified trees, and T_m is used for the merged tree. T' will be used to refer to either of the modified trees.

To illustrate three-way merging of XML we give two examples in the following sections.

2.1 Example 1: Structured Document Merge

Figure 1 shows concurrent edits to a structured document. T_0 is the original document, two copies of which are modified into T_1 and T_2. In T_1, an additional answer is added to the joke, a typo is corrected ("Jokse"→"Jokes"), and the footnote is deleted. In T_2, the structure of the document has been revised so that <sect>[2] becomes a subsection of <sect>[1]. The title of <sect>[2] has also been updated. A merged version integrating the edits from T_1 and T_2 is shown as T_m in the figure.

In particular, we want to emphasize the merging of structure that occurs in the example: although <sect>[2] is moved in T_2, it does not prevent us from making changes to the structure of that subtree in T_1 (in this case inserting a new answer). We feel that such "reorganization in the large" in combination with "updates to the details" is a particularly important XML merge case which tools lacking the notion of moves (e.g. diff and patch) are unable to handle in the general case.

The example is deliberately quite contrived in order to show the integration of as many types of change as possible; in this case an attribute update, as well as insert, delete, and move of XML elements. The edits to T_1 and T_2 would not occur as densely in a more realistic example.

2.2 Example 2: Variants of a Web Page

The second example, shown in figure 2, makes use of three-way merging for maintaining multiple variants of the home page of a fictive chess club. There are two variants of the XHTML [14] home page, one for full-color large-screen

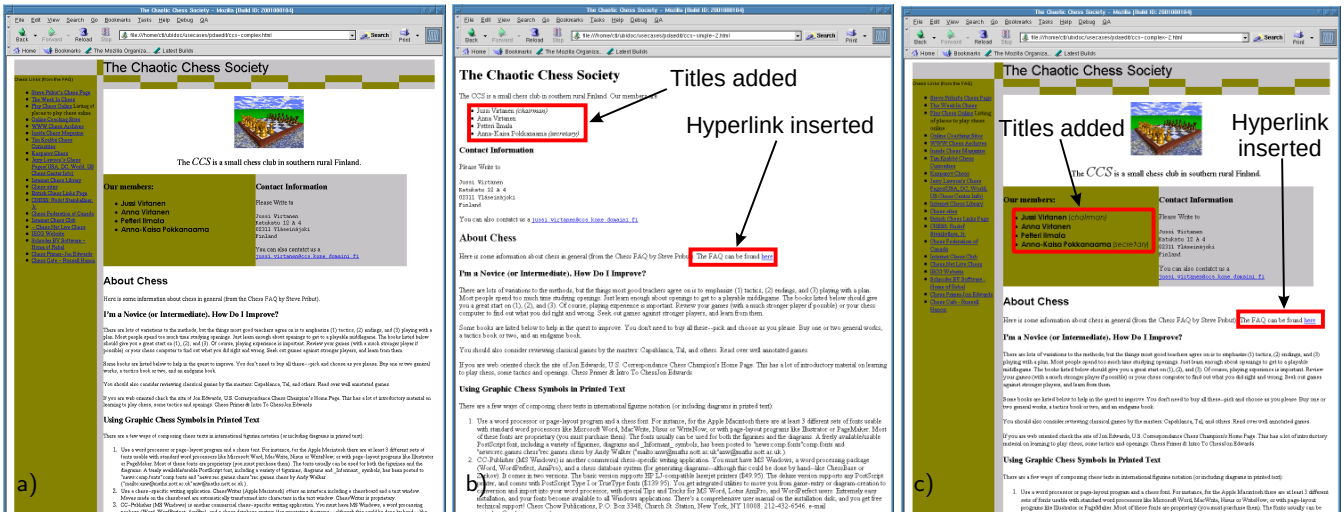

Figure 2: Merging of web page variants. a) full, b) modified simple, and c) full with merged changes

browsers (figure 2a) and another, simple version (figure 2b) for more limited environments, such as a small screen PDA. As can be inferred from the figures, the variants have several XHTML fragments in common (body text, list of links, etc).

We make some changes to the simple version: a hyperlink is included, and titles (chairman, secretary etc.) are added to the names on the member list. We want these changes applied to the full version as well, as shown in figure 2c.

This may be accomplished by three-way merging of the simple (as T_0), full (as T_1), and modified simple (as T_2) variants. In this case, the detected edit set E_1 from T_0 to T_1 would delete everything from T_0 not in T_1 and add everything that is only in T_1 (HTML tables, the image, etc.). The detected edit set E_2 between T_0 and T_2 changes the common parts of T_0 and T_2 (such as the parts we modified), which in turn are untouched by E_1. By combing these, T_0 is transformed into T_1 by E_1, and finalized into the desired merge result by E_2.

Note that it is irrelevant whether the full variant was actually created by editing the simple variant (or vice versa), as the edit detection phase may synthesize edit sets between arbitrary trees. This useful property of three-way merging allows us to merge documents ad hoc without any knowledge of their revision history, or indeed any other information on how they are related.

2.3 XML Ordered Tree Model

We model XML documents as ordered trees with labeled nodes. The node labels are unique among all documents, and hence act as unique identifiers for the nodes. Each tree node has a content which consists of the node type (element or text), character data (in the case of text nodes), and an element name and a set of attributes and values (in the case of element nodes).

The model above leaves out comments, DTDs, entities, processing instructions, and other such constructs in order to simplify the presentation of the merge. We note that the full XML infoset [16] can easily be accommodated for by adding additional node types to the tree model; e.g. to add comments, one would add a comment node type and store the comment in the character data of the node, etc.

As labels we use R for the root and $a, b, \ldots, j$ for any other nodes. Subscripts are optionally used to identify the tree of a label, i.e. R_0 and a_0 are in T_0. Primes ($'$) are used to denote updated content with respect to T_0. The letters $n, \ldots, z$ denote node variables. Furthermore, the relation $\mathrm{c}(n, \mathbf{c})$ denotes that the node labeled n has the content $\mathbf{c}$.

It suits our purposes to express a tree as a set of content and *parent-child-successor* (PCS) relations. The latter is a ternary relation $\mathrm{pcs}(r, p, s)$ expressing that r is the parent of both p and s, and that s immediately follows p in the child list of r.

The set of relations expressing the tree T_k is denoted $\mathbf{T}_k$. To express the start and end of a child list we use the special symbols $\dashv$ and $\vdash$, so that $\dashv$ precedes the first node in the child list, and $\vdash$ succeeds the last node. In addition, we assert an artificial parent $\perp_k$ for the root R_k, as well as the PCS relation for an empty child list for each leaf node, thereby guaranteeing the existence of a relation $\mathrm{pcs}(n, \cdot, \cdot)$ for each node n:

1. $\{\mathrm{pcs}(\perp_k, \dashv, R_k), \mathrm{pcs}(\perp_k, R_k, \vdash)\} \subset \mathbf{T}_k$
2. $\{\mathrm{pcs}(n_k, \dashv, \vdash)\} \subset \mathbf{T}_k$, where n_k is a leaf node in T_k

The union of the relations over all trees is denoted $\mathbf{T} := \cup \mathbf{T}_k$. We use the set membership notation $n \in T_k$ to indicate that a node n exists in T_k.

Consider, for instance, the tree T_0 corresponding to the XML fragment <a>hello. Expressed as PCS and content relations

$$\begin{aligned}\mathbf{T}_0 \quad = \quad & \{\mathrm{pcs}(\perp_0, \dashv, a_0), \mathrm{pcs}(\perp_0, a_0, \vdash), \mathrm{pcs}(a_0, \dashv, b_0), \\ & \mathrm{pcs}(a_0, b_0, \vdash), \mathrm{pcs}(b_0, \dashv, \vdash), \\ & \mathrm{c}(a_0, \{\mathrm{type}: \ \mathrm{element}, \mathrm{name}: \mathsf{a}, \mathrm{attributes}: \{\}\}), \\ & \mathrm{c}(b_0, \{\mathrm{type}: \ \mathrm{text}, \mathrm{chardata}: \mathsf{hello}\})\}\end{aligned}$$

where a_0 and b_0 are the labels of the nodes.

As a shorthand we use the less exact $\mathrm{c}(n, \mathbf{c})$ for the content of a node n that corresponds to the markup $\mathbf{c}$, e.g $\mathrm{c}(a_0, <\mathsf{a}>)$ in the example above.

3. MERGE RULES FROM USE CASES

As a starting point for defining the merge we constructed a set of use cases demonstrating different situations where merging of XML documents is used. Each case included a set of XML input files, merging tasks, and the desired results of the merging task. The desired results were obtained by merging the input files by hand, according to whatever ad-hoc rules generated a meaningful and appropriate result in the context of the use case. The use cases were of two kinds: relatively small and abstract "studies" on XML merging, and rather elaborate "real-world" examples.

The study cases were devised so as to explore different configurations that one may come across when merging XML. They aim to answer such questions as: What is a reasonable course of action when a node was moved in T_1 and updated in T_2? What if a node was moved in T_1 and deleted in T_2? How should the child lists *ab*, *abi*, and *abj* in T_0, T_1, and T_2 be merged (i and j appended in different trees)?

In the study cases, we used various combinations of node update, insert, delete, move, and copy operations to form the modified trees from T_0. For each case, a hand-crafted merged file was constructed to demonstrate a reasonable answer to the merging problem posed by the input files. A total of 37 of these cases were devised, one of which is illustrated in figure 3.

The elaborate cases all originate in a common scenario (or user "story"). We will describe each case briefly here, in order to give the reader an impression of the grounds from which our merge rules are derived.

Concurrent editing of a rich-text document. A technical manual is concurrently edited (text corrections, paragraph moves and deletes) and reformatted (style definitions changed, and styles applied to various parts of the text). The case makes use of the native XML format used in the OpenOffice [11] word processor. Several three-way merge runs (as there are more than two concurrently edited versions) are used to integrate all changes into a unified document.

Updating XML documents that share data. The case consists of three XML documents: a data record with personal information and two XHTML web pages that include markup from the record. The web pages also share some common markup not found in the personal record. Three-way merging is used to propagate changes introduced in one of the documents to the others.

Keeping an inlined SVG drawing up-to-date. The case involves two documents: an SVG [15] drawing and an XHTML document, into which an annotated version of the drawing has been included (so that the SVG markup is part of the XHTML file). Three-way merging is used to propagate updates of the SVG drawing into the manual, without destroying the annotations to the drawing.

Maintaining a shared shopping list. An XML file containing a structured shopping list is concurrently modified in several rounds. Three-way merging is repeatedly used to generate an integrated version of the list.

Maintaining several versions of a web page. The case is similar to the example described in section 2.1.

3.1 Merge Rules

During subsequent analysis of the use cases we identified the following general rules for the merge:

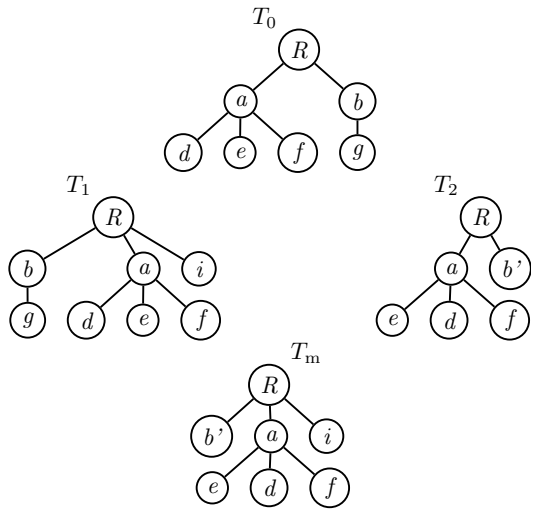

Figure 3: Sample merge case

1. In order to reconcile changes originating in different trees, we need to have a notion of "sameness" between tree nodes. We cannot, for instance, reconcile a move of a node in T_1 with an update in T_2, unless we know they apply to the same node.

2. We define parts of the structure of T_m by requiring it to contain certain node contexts. A *node context* consists of the parent, predecessor, and successor of a node.

3. We also require some node contexts, called *guards*, to appear in T_m in order to inhibit merging of changes originating in different trees that are too close to one another for us to assume semantic independence.

4. Changes to the content of a node in T' should also occur in T_m.

5. A node that is inserted in T' should also appear in T_m. The context of the node in T_m should be the same as its context in T'.

6. A node removed from T' should not appear in T_m. As a guard, we require that the context of the deleted node is retained in T_m. E.g. if d is removed from the child list *adb* of R, then *ab* should appear as a sublist of the child list of R in T_m.

7. Nodes moved in T' should be correspondingly moved in T_m:

   (a) We find that the destination is better thought of as a relative, rather than absolute, position in the tree. For instance, if we reorder paragraphs in a section, we want to retain the reordering *inside* the section, but do not care if the entire section is moved.

   (b) We find (with few exceptions) that a node context is suitable to express the destination of a move: a moved node should occur in T_m in the same context as in T'.

(c) As a guard, we retain the context around the source of the move in T_m. For example, if b is moved from the child list abc of R, then ac should appear as a sublist of the child list of R in T_m.

(d) The exceptions to 7b) represent moves originating in different trees that may be considered "too close" to merge safely. For example, the child lists $abcd$, $bacd$, and $abdc$ in T_0, T_1, and T_2 respectively.

8. Several types of conflicts were identified, e.g. updating the content of a node in both T_1 and T_2.

The node context, and its use to specify parts of T_m is the key result of the analysis, and the cornerstone for the merge. The merge rules on tree structure (5–7) are all statements on preservation of node context in T_m.

To illustrate the use of node contexts, let us consider the merging example in figure 3. In T_1, we see that i was inserted and that b and a have been swapped. By rule 5, we find that i should be included in T_m as a child of R, it should be preceded by a, and should have no successor. Rule 7 yields that the child list of R in T_m should start with the nodes bai (regardless of whether we consider a or b to be the moved node).

In T_2, there are three changes: an update of b, the deletion of g, and the swapping of d and e. Rule 4 tells us that we should use the updated content of b'_2 for b in T_m. The delete implies an empty child list for b (rule 6), and the node swap implies that the child list of a in T_m should start with the nodes edf (rule 7).

We note that there are no other implications on the positions of d, e, and f due to the swap in T_2. In particular, the position of their common parent a is not fixated in T_m due to the swap. This treatment allows us to successfully combine the swap with the new positions for a and b, and construct a merged tree (T_m in figure 3) which contains all contexts implied by the merge rules.

4. TREE MATCHING

To be able to relate the "same" node in different trees (merge rule 1) we use the notion of a *matching* (or *mapping*) (see e.g. [2, 3]) relation between the nodes of the trees. We express matchings with the predicate $\mathrm{m}(n, m)$, which is true if and only if n and m are matched.

A matching between nodes in T_0 and T' may quite naturally be interpreted in terms of edit operations. A node $n \in T_0$ with no match in T' is *deleted* and a node $m \in T'$ with no match in T_0 is *inserted*. A node $m \in T'$ matched to a node $n \in T_0$ is *updated* if the content of n and m differs, and *moved* if the position (according to some definition) of n and m differs. A node $n \in T_0$ that has many matches in T' has been *copied*. While we do not require such an interpretation for the definition of the merge, we find it useful to talk informally about inserted, deleted, etc., nodes when describing the merge.

We require that any node in T' may be matched to at most one node in T_0, and that any node in T_0 may have at most one match in each of T_1 and T_2. We denote this requirement with R_1. Furthermore, we will find it convenient to require that m is an equivalence relation, thereby partitioning the set of tree nodes into a set of disjoint equivalence classes.

R_1 excludes the notion of forming a node in T' by combining several nodes from T_0 (sometimes referred to as the *glue* operation [2]), as well as the notion of copied nodes from our model. Both operations, especially glue, were found to be uncommon and too dependent on the semantics of the data to be included in a general three-way merge. If encountered, these operations are modeled as deletes and inserts.

When defining the merge we will find it convenient to let all $\perp_k$ be matched (requirement R_2). In order for m to be an equivalence relation, it has to be reflexive (R_3), symmetric (R_4), and transitive (R_5).

The requirements are formalized below. Note that R_4 (symmetry) simplifies the expression of R_1.

(R_1) $\forall s, t, u \in \cup T_k : \mathrm{m}(s, t) \wedge \mathrm{m}(s, u) \rightarrow u = t$

(R_2) $\forall n, m \in \cup \{\perp_k\} : \mathrm{m}(n, m)$

(R_3) $\forall n \in \cup T_k : \mathrm{m}(n, n)$

(R_4) $\forall n, m \in \cup T_k : \mathrm{m}(n, m) \rightarrow \mathrm{m}(m, n)$

(R_5) $\forall s, t, u \in \cup T_k : \mathrm{m}(s, t) \wedge \mathrm{m}(t, u) \rightarrow \mathrm{m}(s, u)$

The matching formally defines the notion of node sameness used by the merge, and will normally map to some intuitive and meaningful notion of "same node", e.g. the <sect>[1] elements in figure 1 should be matched to each other.

We assume that a matching relation satisfying R_1–R_5 is available as an input to the merge. Having a matching that accurately tracks the same node in the base and modified trees is important for enabling the trees to be merged. It is, for instance, not possible to integrate the move and content update to b in figure 3, unless we know that b_0, b_1 and b'_2 are the same node, i.e. the nodes need to be matched to each other.

Building the matching usually requires solving some variant of the widely studied *tree matching problem* (see e.g. [2, 3]). In practice, we have constructed the matching by heuristically building one-to-one node matchings between T_1 and T_0, as well as between T_2 and T_0, and then trivially extending the matching to satisfy conditions R_2–R_5. We will not discuss tree matching further in this paper, but do point out that the issue of constructing matchings efficiently and accurately is of large practical importance when implementing XML merging for documents without unique element identifiers.

5. CHANGE MODEL

In order to reconcile changes, we first need to define what a change actually is, and how to express it. We define two types of changes to trees, content and structural, and express these using the same relations we use for expressing the trees themselves.

A *content change* is a relation $\mathrm{c}(n, \mathbf{c})$, denoting that the content of n is changed to $\mathbf{c}$, and a *structural change* is a relation $\mathrm{pcs}(r, p, s)$, denoting that r, p, and s were arranged into a parent-child-successor relationship. The structural change is the formalization we have chosen for the notion of node context (merge rule 2).

Consider the changes $\mathrm{c}(a_0, \text{<a2>})$ and $\mathrm{pcs}(R_0, \dashv, a_0)$, for instance. These express that the content of a_0 is changed to <a2> and that a_0 has been positioned at the start of the child list of R_0.

When combining several changes into a change set, we want to ensure that the set is *consistent*, by which we mean that the changes in the set are unambiguous. This is the case if the changes in the set state no more than one content, parent, predecessor, and successor for each node:

1. $\mathrm{c}(n, \mathbf{c}) \wedge \mathrm{c}(n, \mathbf{c}') \rightarrow \mathbf{c} = \mathbf{c}'$ (Uniq. content)
2. $\mathrm{parent}(r, n) \wedge \mathrm{parent}(r', n) \rightarrow r = r'$ (Uniq. parent)
3. $\mathrm{pcs}(r, p, s) \wedge \mathrm{pcs}(r, p', s) \rightarrow p = p'$ (Uniq. predecessor)
4. $\mathrm{pcs}(r, p, s) \wedge \mathrm{pcs}(r, p, s') \rightarrow s = s'$ (Uniq. successor)

where $\mathrm{parent}(r, n)$, $n \notin \{\dashv, \vdash\}$ denotes that r is the parent of n ($\mathrm{parent}(r, n) \Leftrightarrow \exists\, x : \mathrm{pcs}(r, n, x) \vee \mathrm{pcs}(r, x, n)$). A change δ is *consistent with a change* δ' if and only if the set $\{\delta, \delta'\}$ is consistent.

The change notation offers us an alternative interpretation of the set $\mathbf{T}_k$ (i.e. T_k expressed as PCS and content relations; see section 2.3) as a change set that fully describes how to form T_k from an arbitrary initial state. Conversely, we may use a set of content and structural changes to unambiguously express a tree, a property which will be used to build T_m.

6. DEFINITION OF THE MERGE

For the purpose of the merge, we use the matching to group together the corresponding (i.e. "same") nodes from different trees. To express this grouping we introduce the *node class*, which is the equivalence class of a node under the matching relation m. Each node class has a *class representative*, which is the node in the class that belongs to the tree T_k with the smallest k. For instance, the class representative of $\{n_1, n_2\}$ is n_1 since $n_1 \in T_1$.

To equate the same node from different trees, we operate on node classes rather than the nodes themselves. We denote by $\mathbf{T}^*$ the set where all nodes in the changes in $\mathbf{T}$ have been replaced by the representative of their node class.

Note that the replacement of nodes with their class representatives usually causes a change set to become inconsistent. E.g. if the set of relations expressing the input trees $\mathbf{T} = \{\mathrm{c}(a_0, \mathbf{c}), \mathrm{c}(a_1, \mathbf{c}'), \mathrm{pcs}(a_2, \dashv, b_2), \mathrm{pcs}(a_0, \dashv, b_0), \ldots\}$ then $\mathbf{T}^* = \{\mathrm{c}(a_0, \mathbf{c}), \mathrm{c}(a_0, \mathbf{c}'), \mathrm{pcs}(a_0, \dashv, b_0), \ldots\}$. Here, $\mathbf{T}$ is consistent, but $\mathbf{T}^*$ is inconsistent (as the content of a_0 is inconsistent). Intuitively, $\mathbf{T}^*$ will be the "raw" merge of the trees in $\mathbf{T}$, with inconsistent content and position for some of the nodes.

The purpose of our merge is to deduce the set of changes that express the merged tree, i.e. $\mathbf{T}_\mathrm{m}$, from the sets $\mathbf{T}_0, \mathbf{T}_1$, and $\mathbf{T}_2$. The notion of including certain node contexts and contents from the modified trees into T_m corresponds to including changes from $\mathbf{T}_1$ and $\mathbf{T}_2$ into $\mathbf{T}_\mathrm{m}$.

Unfortunately, we do not know the contexts or updates implied by the merge rules, as the actual edit scripts are unknown. We will thus need to detect the changes that represent these contexts and updates.

We observe that a difference between $\mathbf{T}_1^*$ or $\mathbf{T}_2^*$ and $\mathbf{T}_0^*$ implies that an edit has been made. Although the converse is not true in every case (e.g. edits whose effects on the tree cancel out), it presents a reasonable way to detect edits: a (detected) *edit* is a change in $\mathbf{T}_1^*$ or $\mathbf{T}_2^*$ that is not in $\mathbf{T}_0^*$. Thus, we define the set of edits $\mathbf{E} := \mathbf{T}^* - \mathbf{T}_0^*$. This is the *edit detection* phase.

Let Δ be a consistent set of changes that expresses a forest whose trees are T_m (the root of which will be $\perp_0$) and any deleted subtrees. We want Δ to contain all edits, i.e. all members of $\mathbf{E}$. Furthermore, Δ may only contain changes from the input trees, i.e. no changes may be "made up":

1. Δ is a consistent forest.
2. $\mathbf{T}_\mathrm{m} \subset \Delta$ expresses T_m (whose root is at $\perp_0$)
3. $\mathbf{E} \subset \Delta$
4. $\Delta \subset \mathbf{T}^*$

Conditions 3 and 4 give us $\mathbf{E} \subset \Delta \subset \mathbf{T}^*$. Also, $(\mathbf{T}^* - \mathbf{E}) \subset \mathbf{T}_0^*$. This implies that Δ can only be constructed by removing changes from $\mathbf{T}^*$ that are also in $\mathbf{T}_0^*$, i.e. only content and structure from the base tree may be discarded. This is in concert with our interpretation that the edits in T_1 or T_2 are those that need to be preserved in T_m.

We want Δ to retain as much as possible of the original tree. We accomplish this by starting with $\Delta = \mathbf{T}^*$ and repeatedly removing any change $\delta \in \mathbf{T}_0^* \cap \Delta$ that is inconsistent with some other change $\varepsilon \in \Delta$ until Δ becomes consistent. ε will always be and edit, since it can never be in $\mathbf{T}_0^*$ (as that would imply that $\mathbf{T}_0^* \supset \{\delta, \varepsilon\}$ is inconsistent, which is false). δ, on the other hand, is never an edit, and hence we *preserve all edits in* Δ. This is the *reconciliation* phase of the merge.

A failure to make Δ consistent with this method implies that $\mathbf{E}$ is inconsistent, i.e. there are conflicting edits in T_1 and T_2. Our definition of consistency allows modifications to deleted subtrees, leaving the issue of deciding if concurrent deletes and modifications are conflicts to be externally determined. Unresolved inconsistencies, i.e. conflicts, may however not appear in the deleted subtrees.

Pseudocode for a straightforward implementation of our merge is shown in figure 4. The procedure takes the input trees and a matching relation as arguments. We initialize by converting the trees to sets of structural and content changes, using the matching relation to replace nodes with their class representatives (lines 1–4). We then combine the sets of changes into a "raw" merge, which is the initial value for Δ (line 5). Next, we remove any inconsistencies by iterating over Δ, and looking up any change δ' that is inconsistent with the current change δ, i.e. a change that states another content, root, predecessor or successor than δ (lines 7–11). If an inconsistency is found, we remove the change that originates in the base tree, or signal a conflict if both changes are edits (lines 12–20).

6.1 Edits Introduced by Updating, Inserting, Deleting or Moving a Node

Next, we investigate which edits will be introduced to the merged tree when we perform an update, insert, delete or move of a node in T'. We assume that we make the "first change", i.e. T' and T_0 are identical prior to the change. Note that we may always assume the existence of predecessors and successors of a changed node in the PCS model (since $\dashv$ and $\vdash$ cannot be edited).

When we *update* the content of a node n from $\mathbf{c}$ to $\mathbf{c}'$, we get two inconsistent content changes in $\mathbf{T}^*$ of which $\mathrm{c}(n, \mathbf{c}')$ is the edit. The content of n will thus be $\mathbf{c}'$ in T_m.

Assume we *insert* a node n between s and p below r. The insertion corresponds to the changes $\mathrm{pcs}(r, s, n)$ and $\mathrm{pcs}(r, n, p)$. The changes are not in $\mathbf{T}_0^*$ since $n \notin T_0$, and

```
procedure merge(T0,T1,T2: Tree, m: Matching)
 1: for 0 ≤ k < 3 do
 2:    T:=asChangeSet(Tk)
 3:    T*k:=useClassRepresentative(T, m)
 4: end for
 5: Δ := ∪ T*k { Δ initially "raw" merge }
 6: for all δ ∈ Δ do
 7:    δ':=nil { Holder for inconsistent change }
 8:    δ':=getOtherContent(Δ,δ)
 9:    if δ' = nil then δ':=getOtherRoot(Δ,δ)
10:    if δ' = nil then δ':=getOtherPredecessor(Δ,δ)
11:    if δ' = nil then δ':=getOtherSuccessor(Δ,δ)
12:    if δ' ≠ nil then
13:       if δ' ∈ T*0 then
14:          Δ := Δ − {δ'}
15:       else if δ ∈ T*0 then
16:          Δ := Δ − {δ}
17:       else
18:          conflict(δ, δ') { δ and δ' conflict }
19:       end if
20:    end if
21: end for
22: return Δ
endproc
```

Note: The getOther*(Δ, δ) functions return nil when no maching change other than δ is found in Δ.

Figure 4: Pseudocode for an implementation of our merging definition

hence edits. Thus, the changes guarantee the subsequence $\ldots snp \ldots$ in the child list of r in T_{m}.

If we *delete* d from the subsequence $\ldots pds \ldots$ in the child list of r, we get the change $\mathrm{pcs}(r, p, s)$, which is an edit since $\mathbf{T}_0^*$ states d as predecessor for s and successor for p. Hence, $\mathrm{pcs}(r, p, s)$ occurs in T_{m}.

Finally, we consider *moving* away n from the subsequence $\ldots pns \ldots$ in the child list of r. There are two cases of node move: inter- and intraparent. In the former case, the parent of the node changes to t, corresponding to a pair of delete/insert operations: $\mathrm{pcs}(r, p, s)$, $\mathrm{pcs}(t, u, n)$, and $\mathrm{pcs}(t, n, v)$ $(r \neq t)$. These changes are all edits, and will hence occur in T_{m}.

In the intraparent case the edits are $\mathrm{pcs}(r, p, s)$, $\mathrm{pcs}(r, u, n)$, and $\mathrm{pcs}(r, n, v)$ (provided the node is actually moved), and hence the child list of r in T_{m} will have the subsequences $\ldots ps \ldots$ and $\ldots unv \ldots$. Thus, the "hole" left by the moved node, as well as the neighborhood around the node destination, will be present in the merged tree.

In the discussion above we considered the effect of a single initial node operation to T'. In the case of several operations there may be dependencies between these, making the analysis in terms of changes more involved. The net effect still remains the same, however: whenever a node is deleted, inserted or moved, the node contexts at the locations where the node disappeared and appeared occur in the merged tree as they do in the modified tree.

Finally, we note that there is an easy way to force changes to become edits, which may be useful when we have some particularly important data that we require to be in T_{m} (although it may be identical to the data in T_0). To force a node n to appear in the same context and with the same content in T_{m} as it does in T', we simply do not match it to a node in T_0, causing $\{\mathrm{pcs}(r, p, n), \mathrm{pcs}(r, n, s), \mathrm{c}(n, \mathbf{c}')\}$ to be edits.

6.2 Trace of a Merge

To demonstrate our merge, we apply it to the trees in figure 3. Since listing all members of the various sets would make the example illegible, we explicitly list the content and PCS relations of b only. Furthermore, we assume that the content of b is <b> in T_0, T_1 and <b2> in T_2.

The input trees expressed as change sets are

$$\begin{aligned}
\mathbf{T}_0 &= \{\mathrm{pcs}(b_0, \dashv, g_0), \mathrm{pcs}(b_0, g_0, \vdash), \mathrm{c}(b_0, \texttt{<b>}), \ldots\},\\
\mathbf{T}_1 &= \{\mathrm{pcs}(b_1, \dashv, g_1), \mathrm{pcs}(b_1, g_1, \vdash), \mathrm{c}(b_1, \texttt{<b>}), \ldots\}, \text{ and}\\
\mathbf{T}_2 &= \{\mathrm{pcs}(b_2, \dashv, \vdash), \mathrm{c}(b_2, \texttt{<b2>}), \ldots\}
\end{aligned}$$

The "raw merge" of these is

$$\begin{aligned}
\mathbf{T}^* = \{&\mathrm{pcs}(b_0, \dashv, g_0), \mathrm{pcs}(b_0, g_0, \vdash), \mathrm{pcs}(b_0, \dashv, \vdash),\\
&\mathrm{c}(b_0, \texttt{<b>}), \mathrm{c}(b_0, \texttt{<b2>}), \ldots\}
\end{aligned}$$

which we obtain by replacing the nodes in $\cup \mathbf{T}_k$ with their class representative. We note that this set is inconsistent, e.g. regarding the successor of $\dashv$ in the child list of b.

The edits are the changes not in $\mathbf{T}_0^*$, in this case $\mathbf{E} = \{\mathrm{pcs}(b_0, \dashv, \vdash), \mathrm{c}(b_0, \texttt{<b2>}), \ldots\} = \mathbf{T}^* - \mathbf{T}_0^*$. The changes we need to remove from $\mathbf{T}^*$ to achieve consistency are

$$\{\mathrm{pcs}(b_0, \dashv, g_0), \mathrm{pcs}(b_0, g_0, \vdash), \mathrm{c}(b_0, \texttt{<b>}), \ldots\} \subset \mathbf{T}_0^*,$$

yielding

$$\Delta = \{\mathrm{pcs}(b_0, \dashv, \vdash), \mathrm{c}(b_0, \texttt{<b2>}), \ldots\}$$

Note that $\mathbf{E} \subset \Delta$, i.e. all edits are in the merged set.

T_{m} may be traversed by starting from $\perp_0$ and following the changes in Δ, with $\dashv$ as initial condition for the child list of each node, as shown in figure 5. In the figure, we have used subscripts for the structural changes to show which tree(s) they originated in. For those changes that are edits, we have also shown which changes in $\mathbf{T}_0^*$ the edits are inconsistent with.

6.3 Conflicts

The types of conflicts that are identified is an important aspect of the merge. Ideally, we want the identified conflicts to correspond to our intuitive understanding of what may be merged and what may not. On one hand, this means that the merge should not find conflicts where there does not seem to be any. On the other hand, we do not want a merge that is "too clever" in the sense that it automatically resolves ambiguous situations.

Consequently, several conflict situations were included in the use cases in order to help us to identify a set of intuitively meaningful conflicts. The merge presented here was found to be too tolerant with respect to combinations of edits and deletes to fit the definition of a generalized merge. Fortunately, these additional conflicts are easy to detect, e.g. during post-processing. Separating some of the conflict detection from the actual merging has the advantage of making it optional, providing users with less strict merging, if desired.

There are some ambiguous situations where we may make a reasonable guess regarding the desired outcome of the merge. For instance, if there are node appends to the same

Child list of $\perp$

$\dashv$	Initial condition
$\dashv R$	change $\text{pcs}_{012}(\perp, \dashv, R)$
$\dashv R \vdash$	change $\text{pcs}_{012}(\perp, R, \vdash)$

Child list of R

$\dashv$	Initial condition
$\dashv b$	edit $\text{pcs}_1(R, \dashv, b)$
	inconsistent with $\{\text{pcs}_{02}(R, \dashv, a), \text{pcs}_{02}(R, a, b)\}$
$\dashv ba$	edit $\text{pcs}_1(R, b, a)$
	inconsistent with $\{\text{pcs}_{02}(R, \dashv, a), \text{pcs}_{02}(R, b, \vdash)\}$
$\dashv bai$	edit $\text{pcs}_1(R, a, i)$
	inconsistent with $\{\text{pcs}_{02}(R, a, b)\}$
$\dashv bai \vdash$	edit $\text{pcs}_1(R, i, \vdash)$
	inconsistent with $\{\text{pcs}_{02}(R, b, \vdash)\}$

Child list of b

$\dashv$	Initial condition
$\dashv\vdash$	edit $\text{pcs}_2(b, \dashv, \vdash)$
	inconsistent with $\{\text{pcs}_{01}(b, \dashv, g), \text{pcs}_{01}(b, g, \vdash)\}$

Child list of a

$\dashv$	Initial condition
$\dashv e$	edit $\text{pcs}_2(a, \dashv, e)$
	inconsistent with $\{\text{pcs}_{01}(a, \dashv, d), \text{pcs}_{01}(a, d, e)\}$
$\dashv ed$	edit $\text{pcs}_2(a, e, d)$
	inconsistent with $\{\text{pcs}_{01}(a, \dashv, d), \text{pcs}_{01}(a, e, f)\}$
$\dashv edf$	edit $\text{pcs}_2(a, d, f)$
	inconsistent with $\{\text{pcs}_{01}(a, d, e), \text{pcs}_{01}(a, e, f)\}$
$\dashv edf \vdash$	change $\text{pcs}_{012}(a, f, \vdash)$

Child list of e

$\dashv$	Initial condition
$\dashv\vdash$	change $\text{pcs}_{012}(e, \dashv, \vdash)$

Child list of d, f similar to e

Child list of i

$\dashv$	Initial condition
$\dashv\vdash$	edit $\text{pcs}_1(i, \dashv, \vdash)$
	inconsistent with $\{\}$

Content of $b{=}b'_2$

Content of R, a, d, e, f, i same as in T_0

Figure 5: Merge of the trees in figure 3

child list that originate in different trees, we may guess that the merged child list should contain both appends. Our implementation includes, as an option, functionality for such speculative merging.

There are two categories of conflicts: core (C), which are identified by a failure of our merge to construct a consistent set Δ, and optional (O), which need to be separately identified. We list the identified conflicts and their category below.

Update/Update (C) The content of a node n is changed in both T_1 and T_2, i.e. $\{\text{c}(n_1, \mathbf{c}_1), \text{c}(n_2, \mathbf{c}_2)\}$ with $\mathbf{c}_1 \neq \mathbf{c}_2$ is an inconsistent subset of $\mathbf{T}^*$.

Position/Position (C) Two nodes $m \in T_1$ and $n \in T_2$ have been positioned so that Δ cannot be made consistent. The positional change of m and n may be a move, insert or delete, yielding the subcategories move/move, move/insert, move/delete, and delete/delete for these conflicts. Our implementation provides speculative resolutions for several of these, e.g. by ignoring the repositioning from either T_1 or T_2.

Delete/Edit (O) The basic merge does not consider edits in any of the deleted subtrees as a conflict. Such a policy may be considered inappropriate, since it loses edits. To guard against such conflicts, we check that $(\Delta - \mathbf{T}_\text{m}) \cap \mathbf{E} = \emptyset$, i.e. that there are no edits among the deleted changes.

6.4 Properties of the Merge

The merge has several desirable properties, which we state briefly.

Symmetry The merge result does not depend on the order in which the modified trees are assigned to T_1 and T_2. This follows from the fact that our merge is indifferent regarding which modified tree a node occurs in; only the distinction between base and modified tree is used.[1]

Preservation of edits All edits are preserved in Δ, since $\mathbf{E} \subset \Delta$. Furthermore, if we consider edits in deleted subtrees as conflicts, $\mathbf{E} \subset \mathbf{T}_\text{m}$, i.e. all edits will be in T_m.

Parallel edit model The detected edits are inherently unordered and independent (in the scope of a single tree) of each other, avoiding the need for any artificial ordering of edits and reconciliation complexities due to that ordering.

Extensibility The formal model presented here has been extended to handle copies in subsequent work. The model should also easily generalize to an n-way merge.

6.5 Computational Complexity

The straightforward merge algorithm shown in figure 4 is implementable in $O(n \log n)$ time, where n is the total number of nodes in the input trees, if we assume that the content size of an individual node is limited by some constant c (this assumption avoids string comparisons etc. from costing more than $O(1)$).

To justify this claim we perform a cursory complexity analysis. As a conceptual aid we use red-black trees [5] which provide $O(\log n)$ lookup and insertion time for pairs of (name, value).

To quickly locate inconsistencies we build lookup tables for the structural changes that are indexed by the predecessor and successor, e.g. the predecessor lookup table could return $\{\text{pcs}(R_0, a_0, \vdash), \text{pcs}(R_0, a_0, i_2)\}$ on the lookup of a_0. Note that the lookup result consists of maximally 3 changes (one for each tree).

The initialization steps of the algorithm (lines 1–4) consists of tree traversal and looking up matched nodes. If we build the lookup tables at the same time we need to do

[1]The use of class representatives that depend on the tree of a node does not break this property. The merge operates on classes, for which the representative is just a convenient label.

$O(\log n)$ work at each node. Finding node matches is constant work, as each node class is limited to 3 nodes (one from each tree). Summing over all nodes, we get $O(n \log n)$.

Combining the sets (line 5) yields $O(n \log n)$ by looping over the nodes and using the indexes to detect duplicates.

The resolving of inconsistencies (lines 6–21) is likewise $O(n \log n)$. Each visited change (of which there are $O(n)$) requires only $O(\log n)$ time to lookup inconsistent rules when we use the lookup tables. getOtherRoot, for instance, finds the relevant changes by looking up the given node in both the predecessor and successor tables.

Finally, we note that the number of edits e will affect the running time, as e.g. the initial size of Δ increases with e. However, the number of edits is maximally $O(n)$ since $|\mathbf{E}| < |\mathbf{T}^*| = O(n)$.

7. IMPLEMENTATION AND EVALUATION

The merge has been implemented as the "3dm" tool for three-way merging of XML, which is developed in an open-source manner at http://tdm.berlios.de. In addition to the three-way merging described here, the tool includes a heuristic tree matcher for building the matching between the base and modified trees.[2]

The implementation was used to validate the practical usefulness of our merge against the use cases, which are also available at the 3dm website. Out of 37 study cases 35 were successful (omitting those parts of the cases demonstrating copy operations). The remaining two cases did not fit our merging model.

In the test runs the output was accepted if it was the hand-merged output of the case, or a reasonable variation thereof. For instance, when appending nodes to a list with appends originating in different trees, it is reasonable to accept items appended in either order. The symmetry of the implementation was also verified.

The merging tasks in the elaborate cases identified some areas of improvement in practical use. One issue was document metadata which will change, usually in an inconsistent manner, in both T_1 and T_2 (e.g. <meta:date> in the OpenOffice files).

Another discovery was that a successful merge requires the trees to have a certain amount of similar structure in common around the areas of change. One example of this is partial matches between text nodes, e.g. a paragraph The number... 555-1234... will not merge with a change to <phone>555-1234</phone>.

Another example where structural edits were too close was an instance of formatting and editing XHTML list items where $T_0 = \ldots$
text
$\ldots$, $T_1 = \ldots$
text
$\ldots$ and $T_2 = \ldots$
text ital
$\ldots$ Here, the merge fails since the text and the italic block are treated as two different nodes, rather than a single logical entity. The case could be fixed by introducing the tag for grouping of such logical entities, e.g. $T_0 = \ldots$ text $\ldots$ and $T_2 = \ldots$ textital $\ldots$

These issues aside, the overall performance of the merge was quite satisfactory, successfully merging the content of several concurrently edited OpenOffice, XML, and XHTML documents. The merge is also being used for reconciliation of directory trees in a shared file system being developed by the author.

[2]The study cases were designed so that the matcher always matched the nodes "correctly", i.e. as intended in the case. There were some "bad" matches in the elaborate cases, but these did not affect the outcome of the merge.

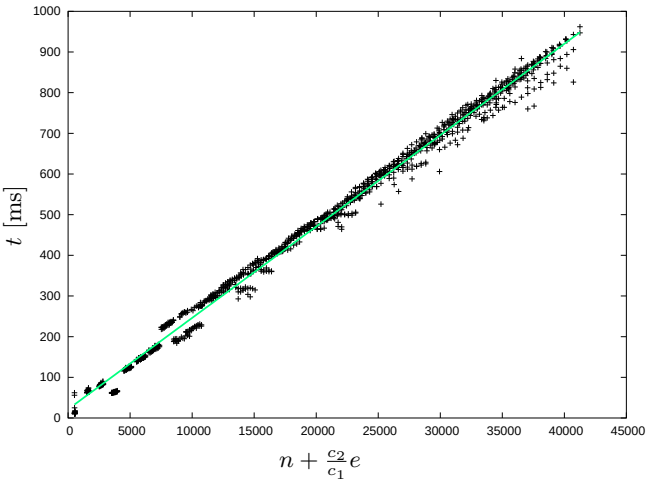

Figure 6: 3dm merge time t versus tree size n and edits e. The time for XML parsing and matching of trees is not included.

The scalability of the 3dm merge implementation was evaluated by timing the merge of a set of trees which varied in number of nodes as well as number of edits. The trees were generated so as to be mergeable without conflicts, and with an equal probability for insert, delete, move, and update operations. Inspection of the results indicated a linear relationship between execution time and the number of nodes as well as the number of edits (hash tables were used rather than red-black trees, as these are often $O(1)$ in practice). Based on this observation, we performed an MSE fit of the data to the linear model $t = c_0 + c_1 n + c_2 e$.

Figure 6 shows a diagram of the execution time t versus the number of nodes n and edits e, with the correlation factor between n and e determined by the fitted model. Visual inspection of the diagram indicates that the model predicts the execution time quite well, as the observations are clearly clustered around the line $t = c_0 + c_1(n + \frac{c_2}{c_1}e)$. The jumps below 15000 nodes appear to be related to the used Java VM configuration (more specifically, the -Xms flag).

8. RELATED WORK

Word processors such as Microsoft Word and, more recently, OpenOffice/StarOffice include integrated tools for document merging. These are, however, based on pairwise comparison of documents, and do not use three-way merging for document reintegration.

In [12] a method for merging hierarchically structured documents and identifying conflicts is outlined. In addition to the base and branch trees the method uses explicit edit scripts, i.e. edit detection is not an integral part of the merge. The edit operations allowed in the script are node insert, delete, and update. Compared to our merge, the method lacks the ability to successfully integrate moves with updates.

An SGML/XML merging algorithm designed for use on technical documentation that is able to integrate an arbi-

trary number of documents into one is presented in [8]. The merged document can in broad terms be described as a level-by-level union of the source trees. The operation of the algorithm can however not be understood as the integration of *changes* with respect to a base version into a merged version, and can thus not be described as a three-way merge.

The DeltaXML [7] tool supports three-way merge of XML. It is possible to do automatic reconciliation of deletions, updates, and inserts. In contrast to our merge, there is no support for the move operation. The tool includes a general tree matching algorithm based on element identifiers and on performing the longest common subsequence alignment at each level of the input trees.

There are several descriptions of three-way merges outside the domain of structured documents that operate on tree-like structures. Merging methods for software (source code), and among them three-way merging techniques are surveyed in [9]. Our merge can be characterized as being state-based, syntactic, and three-way in the terminology of the survey.

A method for structural (hierarchical) merge of software that relies on the change history being available is described in [17]. In [4] Horwitz et al. present a three-way merging algorithm for reconciliation of a restricted class of computer programs, which very well illustrates the complexities of designing a merge for semantically complex data structures. [1] describes a framework for file system synchronization, which entails three-way merging of file systems.

Inserts, deletes, and updates are essentially treated in the same way in all three-way merging methods that the author is aware of. The difference lies in the support for the notion of a "move", and the rules for merging these. Move rules are usually specific to the domain of the merge (e.g. semantically aware source code merging), while we in this work have attempted to find more universally usable rules.

9. CONCLUSIONS AND FUTURE WORK

The examples, use cases and merge rules should provide useful data sets and guidelines for designing and testing three-way merges of document-oriented XML.

We think that the merge presented here, designed according to these guidelines, will prove quite useful from a practical point of view. We motivate this with operation based on empirical rules, successful application to the merging use cases, ease of implementation, computational efficiency, and intuitive handling of node operations. Especially important is the ability to merge "reorganization in the large" with "updates to the details", as exemplified in the use cases. This is also the main advantage of our merge compared to text-based merging and other previous work.

In the future, we plan to extend the merge to handle unordered trees. We will also investigate how the merge can be specialized with domain-specific knowledge. The general idea is to be able to create domain-specific merges that effectively re-use the functionality provided by a generalized merge.

10. REFERENCES

[1] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, pages 98–108.

[2] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press, 1997.

[3] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 493–504, 1996.

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

[5] Guibas L. J. and Sedgewick R. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978.

[6] Michael K. Johnson. Diff, patch, and friends. *Linux Journal*, 1996(28es), August 1996.

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

[8] G. W. Manger. A generic algorithm for merging SGML/XML-instances. In *Proceedings of XML Europe 2001*, Berlin, Germany.

[9] Tom Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[10] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[11] Sun Microsystems. *OpenOffice.org XML File Format 1.0*, December 2002. `http://xml.openoffice.org/xml_specification.pdf`.

[12] Asklund U. Identifying conflicts during structural merge. In *Proceedings of the Nordic Workshop on Programming Environment Research '94*, pages 231–242, Lund, Sweden, June 1994.

[13] W3C. *Extensible Markup Language (XML) 1.0*, 3rd edition, October 2000. `http://www.w3.org/TR/REC-xml/`.

[14] W3C. *XHTML 1.0 Specification*, January 2000. `http://www.w3.org/TR/xhtml1/`.

[15] W3C. *Scalable Vector Graphics (SVG) 1.1 Specification*, January 2003. `http://www.w3.org/TR/SVG/`.

[16] W3C. *XML Information Set*, 2nd edition, February 2004. `http://www.w3.org/TR/xml-infoset/`.

[17] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 68–79. ACM Press, 1991.