A FAULT-TOLERANT THREE-WAY MERGE FOR XML AND HTML

Tancred Lindholm and Torsten Rüger
Helsinki Institute for Information Technology
PO Box 9800
FIN-02015 Helsinki, Finland
email: {tancred.lindholm,torsten.rueger}@hiit.fi

Abstract

Three-way merging is a technique that is used to reintegrate changes to a document when multiple independently modified copies have been made. Tools for three-way merge of ASCII text files exist in the form of the ubiquitous diff and patch tools, but these are of limited applicability when parts of the documents have been rearranged.

Our fault-tolerant three-way merge for XML and HTML was designed to support rearrangements to document structure, as well as situations where the well-formedness of the document has been lost. This is achieved by taking a text-based approach that recognizes moved text and uses normalization and denormalization of whitespace.

As there are many possibilities for merging moved text we decided to base the design of the merge on principles derived from a set of use cases, which systematically explore different merging situations. This design process should help ensure that the chosen merge is useful from a practical point of view.

Keywords Web and Internet Tools, Data Modelling, Collaborative Systems and Applications

1 Introduction

One of the fundamental scenarios in computer-supported collaborative work is the parallel modification of copies of a document, and the subsequent reconciliation of the copies into a single document [1]. We investigate the variant where two replicas of a *base* document have been made and where the replicas have been independently edited, thereby creating two *modified* variants of the base document.

We consider reconciliation based on the technique of *three-way merging* in this paper. The technique requires the base document to be present during reintegration, 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 format of the data it is designed for. In this paper we focus on collaborative work on documents in the XML [2] and HTML [3] formats. We pay particular attention to the cases where the input documents may be syntactically invalid. While this is undesirable, syntactically invalid XML and especially HTML is unfortunately all too common. Also, none of the XML/HTML merging tools that we are aware of currently address the issue of merging malformed input.

As XML/HTML are text-based formats, we could use the ubiquitous Unix diff and patch tools for merging. These are, however, limited due to their insert/delete-based edit model. Even simple restructuring of XML (such as moving a section up one level) may generate huge differences in terms of inserts and deletes, causing diff and patch to break down. These tools are thus not generally suitable for merging changes to document structure.

We approached the design of our merge through a set of use cases. The use cases helped us decide on a merging definition that is useful from a practical point of view. Our key observation was that there were few cases which required knowledge of the hierarchical structure of the input documents in order to merge properly.

We propose a merge tool that goes beyond diff and patch in the sense that it supports merging of restructured documents. In accordance with the findings from the use cases we decided to use a text rather than tree model for the input data. A natural consequence of this decision is resilience to syntactically invalid input.

We start by reviewing related work in section 2. We then introduce definitions and a detailed example of three-way merging of malformed HTML in section 3. In section 4 we present the use cases and the design principles derived from these. The principles are implemented in our merge, which we describe in section 5. We evaluate the merge in section 6, and present conclusions in section 7.

2 Related Work

Related work is found among text alignment and three-way merging algorithms. Much work has been done in the area of finding an optimized alignment of two character strings. The algorithm due to [4] implemented in the diff tool, which aligns by solving the longest common subsequence problem, is perhaps the most well-know example. The differencing algorithm used in the XDelta [5] tool, which is based on the use of hashes and greedy matching of text blocks is also noteworthy for being both simple and computationally inexpensive.

The diff3 [6] tool combines the diff and patch tools to perform three-way merging of text files. The main restric-

tion when applied to merging structurally modified documents comes from the fact that moves are expressed as insert/delete pairs. Due to this, diff3 fails in some situations even though the changes are intuitively mergeable.

The DeltaXML [7] tool supports three-way merge of XML and handles delete, update, and insert operations. There is, however, no support for the move operation. The tool uses a tree matching algorithm based on performing longest common subsequence alignment of element identifiers at each level of the input trees.

Another three-way merging tool for XML is 3dm [8] (`http://tdm.berlios.de`) by one of the authors. The tool supports merging of inserts, deletes, and updates as well as moves. There are some 40 smaller merge cases and several larger tests cases on XML merging available on the 3dm website, which we have utilized in the testing of our merge. Unlike the merge of this paper, 3dm requires that its input is well-formed XML.

There are descriptions of hierarchical three-way merges outside the domain of documents. Merging of software source code, including three-way merging, is surveyed in [9]. Our merge can be characterized as state-based and textual with some syntactic components in the terminology of the survey. In [10] Horwitz et al. present a three-way merging algorithm for reconciliation of a restricted class of computer programs, which very well illustrates the intricacies of designing a merge for complex data structures. [1] describes a framework for file system synchronization, which entails three-way merging of file systems.

3 Three-way Merging

Assume that we have two initially identical replicas of an document D_0 which have subsequently been independently edited, creating the documents D_1 and D_2. Let the *edit scripts* (e.g. [11]) $\mathcal{E}_1$ and $\mathcal{E}_2$ be the ordered sequences of edit operations that created D_1 and D_2 from D_0.

We informally define the result of the three-way merge of D_0, D_1, and D_2 to be the document which is obtained by applying $\mathcal{E}_1$ and $\mathcal{E}_2$ to D_0. We denote the merged object D_m. The length of D_0 is denoted n.

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 D_0 during the reconciliation phase. If edit detection is well implemented, the result of the reconciliation will be the same as if made using $\mathcal{E}_1$ and $\mathcal{E}_2$.

3.1 Example Merge Case

An example case of concurrently edited XHTML fragments is shown in figure 1. The base document D_0 has been mangled in transport (e.g. careless copy-paste from a mail reader), and is no longer well-formed, as the closing angle bracket for the ending <div> tag is missing.

D_1 and D_2 are two edited versions of D_0. In D_1 the second paragraph has been shortened ("into you" is deleted), and the well-formedness has been restored by inserting a ">" at the end. In D_2, the order of the paragraphs is reversed, and the word "my" has been replaced by "his".

In D_m we see how the changes from D_1 and D_2 may be merged: the document is well-formed and the paragraphs are reordered, the first one being shortened and the second one containing "his" instead of "my". The case shows merging of a move, an update, an insert, and a delete.

4 Merge Cases

Our goal was to create a tool that sees its input as plain text, but is also able to take advantage of recurring edit patterns from XML/HTML in order to be able to merge a broader array of situations than the traditional diff and patch tools. As a starting point for defining the merge we constructed a set of small "study" cases demonstrating different situations of XML merging.

Each case included a set of XML input files and the desired result of the merging. The desired result was 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 case. We also used the publically available test cases for the 3dm merging tool.

The study cases were devised so as to explore different configurations that one may come across when merging. They aim to answer such questions as: What is a reasonable action when an XML/HTML fragment was moved in D_1 and updated in D_2? What if a fragment was moved in D_1 and deleted in D_2? How should co-located inserts originating in different documents be handled?

In the study cases, we quite systematically examined combinations of insert, delete, update, and move operations on text, tags, and attributes. Compared to the similar 3dm "small" cases, our cases were more verbose and less self-similar. This caused the average distance between the edits to be significantly higher. In our opinion, this difference makes our cases model actual merging situations more accurately. All in all 97 such cases were written.

4.1 Merge Design Principles

By analyzing the study cases we identified the following general principles for the merge:

1. It is useful to break up the base document into regions of text (text blocks), and think of the modified documents as being constituted of these regions. Each region either has a matching region in D_0, or is a region of inserted text.

2. When the distance between edits is large enough, and the text between the edits is distinct from text else-

```
D1: <div><p>That echoes my thoughts</p><p>So nice to run</p></div>
D0: <div><p>That echoes my thoughts</p><p>So nice to run into you</p></div
D2: <div><p>So nice to run into you</p><p>That echoes his thoughts</p></div
Dm: <div><p>So nice to run</p><p>That echoes his thoughts</p></div>
```

Figure 1. Sample merge case.

where in the document, fairly accurate alignment of the text is possible by doing simple greedy matching, rather than to use minimum edit distance algorithms. This assumption is not true for the "small" 3dm cases, as they are not as verbose as our study cases.

3. Inserted text in D_1 or D_2 should also appear in D_m.

4. Deletes show up as text regions from D_0 that do not appear in either D_1 or D_2. These regions should not be part of D_m.

5. Text moves show up as reordering of text regions in D_1 and D_2 with respect to D_0. Reordered regions should appear in the changed order in D_m.

We found it better to define changed order in relation to the previous and next regions, rather than in terms of absolute position. This allows for merging of overlapping moves, where the latter move touches a subregion of the former.

Consider the situation $D_0 =$ abcde, $D_1 =$ acbde and $D_2 =$ eabcd, where the letters correspond to text regions. If we look at the previous and next regions for determining order, we get the changed ordering acbd from D_1 and ea from D_2, which can both be combined into $D_\mathrm{m} =$ eacbd. In terms of absolute positions, such a merge is not possible.

5 Our Merge Method

The two main phases of the algorithm are matching and merging. In between matching and merging we use the matched documents to build edit scripts, which are used as the input of the merge.

5.1 Matching

To be able to relate the same regions of text in the different documents we build a *matching* between the text in each of the modified documents and the base document. The matching of text between the base (D_0) and a modified document D_i may quite naturally be interpreted in terms of changes to the base document: Text in D_i without a match in D_0 has been *inserted* and text in D_0 without a match in D_i has been *deleted.* Text in D_i that has a match in D_0 is *moved* if its position (according to some definition) differs from the position of the matching text in D_0.

Text in D_0 that has many matches in D_i is said to be *copied* and text in D_i that has many matches in D_0 is said to be *glued* [11]. We chose not to support the copy or glue operations, as these are quite uncommon and merging of these is more often dependent on the semantics of the data. Hence we require the matching to be one-to-one, i.e. each character in D_0 may match at most one character in each of D_i and each character in D_i may match at most one character in D_0.

In a *perfect matching* the characters are aligned according to their actual correspondences in the texts (as determined by e.g. the actual edit operations). While prefect matches are often unobtainable, we only require a *good matching*. A good matching (in the context of a particular merging algorithm) is a matching that yields the same merge result as if the matching had been perfect.

The result of the matching process is an expression of a document as a *match list*. Let M_i be the match list for D_i so that the list expresses D_i using elements of the form $\mathrm{m}(c, l)$ and $\mathrm{ins}(t)$. The former indicates a region of text of length l matching D_0 from the zero-based offset c, and the latter indicates the insertion of the text t. We do not allow consecutive $\mathrm{ins}(\cdot)$ elements in the list, as these may be expressed as one combined insert.

To compute M_1 and M_2 we used a method where we split the document D_i into overlapping regions of size l (covering offsets $0 \ldots l-1$, $1 \ldots l$, etc.) and store hashes for these in a table C_i. Then, going through all hashes in C_0, we look for the first equal one in C_i. When a match is found we expand the region in both directions as much as possible without overlapping with any other matched region in C_0.

The above method guarantees that no text has more than one match in D_i (as we search for *one* region from D_0 in D_i). To ensure there are no glues we check that each character in D_i is matched to at most one in D_0, and remove all but one match in case of multiple matches.

The algorithm depends on the region size l. In order to find unique matches we begin with rather large regions, and then successively decrease the size to find shorter matches. To support text formats better we may also align the start and end of regions on whitespace, thus taking advantage of the natural use of whitespace as delimiters for logical blocks of text (e.g. words in a paragraph).

The matching method easily lends to format-aware enhancements. Using simple lexical scanning, we could for instance make sure that tags and text are never matched, increasing the likelihood of a good matching.

Turning to the sample case of figure 1, we may now construct match lists M_1 and M_2 that express good matchings for D_1 and D_2. Two such match lists are illustrated

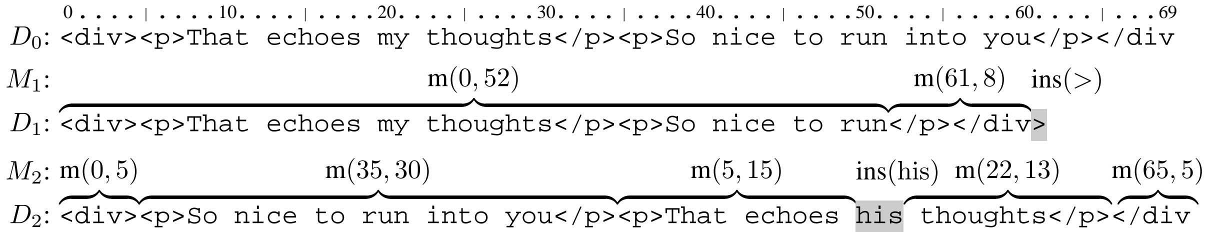

Figure 2. The match lists for D_1 and D_2. Matching regions are marked with $\mathrm{m}(c, l)$ and inserted text like this.

in figure 2. As a practical matter we find it useful to also construct the trivial match list $M_0 = \{\mathrm{m}(0, n)\}$ for D_0.

5.2 From Match Lists to Edit Scripts

The next step is the edit detection phase of the merge, where we convert the match lists for D_1 and D_2 into edit scripts. We start by making an observation regarding regions from D_0 that are matched in both D_1 and D_2. Since such a region is identical in all tree documents, there is no change, and hence the region will either appear unchanged in the merged document, or not at all. That is, we may treat such regions as atomic entities in the merge algorithm.

Unfortunately, we cannot use the match lists of the previous phase directly, since the boundaries of the matching regions are not aligned. For instance, consider the region $\mathrm{m}(35, 17)$. It is shared by both D_1 and D_2 in figure 2; this is however not evident from the match lists.

To overcome this problem we repeatedly split any two matching regions from M_1 and M_2 that overlap into non-overlapping and identical regions, and then replace the original regions in M_1 and M_2 with the appropriate subset of these, until no further overlapping regions exist. The result is that each region in M_1 and M_2 is either identical in both lists, or is completely absent from the other list.

Consider the match lists in figure 2. We find that e.g. $\mathrm{m}(0, 52) \in M_1$ and $\mathrm{m}(22, 13) \in M_2$ overlap, so these are split into $\mathrm{m}(0, 22)$, $\mathrm{m}(22, 13)$, and $\mathrm{m}(35, 17)$. We then replace $\mathrm{m}(0, 52) \in M_1$ with the list $\{\mathrm{m}(0, 22), \mathrm{m}(22, 13), \mathrm{m}(35, 17)\}$. M_2 remains unchanged since it already contains the appropriate subset $\{\mathrm{m}(22, 13)\}$. Repeating this process, we arrive at the regions shown in figure 3a. Note that each matching region may now be uniquely identified by its offset into D_0.

The edit operations we use to express D_1 and D_2 are closely tied to the information available in the match lists, i.e. which parts of D_1 and D_2 match the base document, which are absent, and what text regions have been inserted. The operations are:

Move+Insert which appends a matching region and a block of inserted text to the document. We denote the operation $\mathrm{movi}(c, t)$, where c is the offset of a matching region (which suffices to uniquely identify the region) and t is the text to insert. The text to insert may be empty, in which case we write $\mathrm{movi}(c)$.

Delete+Insert which indicates that a matching region was deleted from the document, and that it was followed by inserted text. The operation is denoted $\mathrm{deli}(c, t)$, where c is the offset of a matching region and t is the text to insert. If no text is inserted, we write $\mathrm{deli}(c)$.

The edit script for D_i, denoted L_i, is an ordered list of these edit operations. We denote with $L_i[c]$ the operation in the list for which the region offset is c, and with $\mathrm{succ}(L_i[c])$ the operation succeeding $L_i[c]$. We build the edit script so that $L_i[c]$ will address exactly one operation for each c.

To generate the edit script from the match list M_i we start with the empty script $L_i = \{\}$ and first loop over M_i to append the $\mathrm{movi}()$ operations, and then over M_0 to insert the $\mathrm{deli}()$ operations.

The first loop traverses M_i appending $\mathrm{movi}(c, t)$ when encountering a pair of $\{\mathrm{m}(c, \cdot), \mathrm{ins(t)}\}$ operations, and just $\mathrm{movi}(c)$ otherwise. The second loop traverses M_0 inserting $\mathrm{deli}(c)$ after $L_i[p]$ for each region c not present in M_i, where $\mathrm{m}(p, \cdot)$ precedes $\mathrm{m}(c, \cdot)$ in M_0. That is, the second loop adds unmatched (deleted) regions from M_0 in the original (M_0) order after the regions in M_i, whose match in M_0 is followed by an unmatched (deleted) region. To account for the document boundaries we use the index $c =$ -1 to signify the start of the document and the index $c = n$ (the length of D_0 being n) to signify the end of the document.

The resulting scripts L_0, L_1, and L_2 will have exactly one operation for each of the matching regions in M_0. This property simplifies the merge algorithm, and is a direct result of the somewhat unorthodox choice of edit operations. It also ensures the existence and uniqueness of $L_i[c]$ for all matching regions $\mathrm{m}(c, \cdot)$ and all i. Figure 3b shows the result of the the edit script generation on the non-overlapping match lists in figure 3a.

Note that our choice of operations allows two ways of expressing adjacent deletes and inserts (i.e. updates) as we may attach the insert to either the delete or the succeeding move. We chose the latter based on experience from the use cases, but observe that both options work well.

5.3 Merging

In the merge phase we construct a merged edit script using the edit scripts for the input documents. We want the merged edit script to retain all changes introduced in the modified documents, and therefore we build the merged

a)

D_0 =<div>|<p>That echoes |my| thoughts</p>|<p>So nice to run| into you|</p>|</div

$M_0 = \{\mathrm{m}(0,5), \mathrm{m}(5,15), \mathrm{m}(20,2), \mathrm{m}(22,13), \mathrm{m}(35,17), \mathrm{m}(52,9), \mathrm{m}(61,4), \mathrm{m}(65,5)\}$

$M_1 = \{\mathrm{m}(0,5), \mathrm{m}(5,15), \mathrm{m}(20,2), \mathrm{m}(22,13), \mathrm{m}(35,17), \mathrm{m}(61,4), \mathrm{m}(65,5), \mathrm{ins}(>)\}$

$M_2 = \{\mathrm{m}(0,5), \mathrm{m}(35,17), \mathrm{m}(52,9), \mathrm{m}(61,4), \mathrm{m}(5,15), \mathrm{ins}(\mathrm{his}), \mathrm{m}(22,13), \mathrm{m}(65,5)\}$

b)

$L_0 = \{\mathrm{movi}(\text{-1}), \mathrm{movi}(0), \mathrm{movi}(5), \mathrm{movi}(20), \mathrm{movi}(22), \mathrm{movi}(35), \mathrm{movi}(52), \mathrm{movi}(61), \mathrm{movi}(65), \mathrm{movi}(n)\}$

$L_1 = \{\mathrm{movi}(\text{-1}), \mathrm{movi}(0), \mathrm{movi}(5), \mathrm{movi}(20), \mathrm{movi}(22), \mathrm{movi}(35), \mathrm{deli}(52), \mathrm{movi}(61), \mathrm{movi}(65, \text{'>'}), \mathrm{movi}(n)\}$

$L_2 = \{\mathrm{movi}(\text{-1}), \mathrm{movi}(0), \mathrm{movi}(35), \mathrm{movi}(51), \mathrm{movi}(61), \mathrm{movi}(5, \text{'his'}), \mathrm{deli}(20), \mathrm{movi}(22), \mathrm{movi}(65), \mathrm{movi}(n)\}$

Figure 3. Non-overlapping match lists and their edit scripts. The region boundaries in D_0 are shown using the | sign.

$L_\mathrm{m} = \{\mathrm{movi}(\text{-1}), \mathrm{movi}(0), \mathrm{movi}(35), \mathrm{deli}(52), \mathrm{movi}(61), \mathrm{movi}(5, \mathrm{his}), \mathrm{deli}(20), \mathrm{movi}(22), \mathrm{movi}(65, >), \mathrm{movi}(\text{-1})\}$

D_m=<div>|<p>So nice to run|</p>|<p>That echoes |his| thoughts</p>|</div|>

Figure 4. The merged document L_m and its textual representation D_m.

reordered(p, s_1)	reordered(p, s_2)	Next position
no	no	$s_1(=s_2)$
yes	no	s_1
no	yes	s_2
yes	yes	conflict

Table 1. Determining next position

$L_1[p]$	$L_2[p]$	$L_1[p'])$	$L_2[p']$	Merge
movi()	movi()	any	any	movi$(p, t_1 t_2)$
deli()	any	movi()	movi()	deli$(p, t_1 t_2)$
deli()	any	movi()	deli()	deli(p)
deli()	any	deli()	any	deli(p)

Table 2. Merging operations. t_1 and t_2 denote the inserted text at $L_1[p]$ and $L_2[p]$ and $t_1 t_2$ their concatenation.

script by always choosing an operation that represents a change from the base document.

Let p indicate the current region in L_0, L_1 and L_2. It is initially set to the start region -1. To determine when a region of text has been moved with respect to D_0 we introduce the predicate reordered(c, s) which is true if, and only if, s is the region of the operation $\mathrm{succ}(L_0[c])$, i.e. if s is the region succeeding c in L_0.

To get the next region p', let s_1 be the region of $\mathrm{succ}(L_1[p])$ and s_2 be the region of $\mathrm{succ}(L_2[p])$. If reordered(p, s_1), then the next region in L_1 is reordered with respect to L_0. In this case, we choose $p' = s_1$, i.e. the next region to output is the next region in L_1. Similarly, if reordered(p, s_2), then $p' = s_2$, i.e. we use the region order from L_2. If the next region is not reordered in any of the modified documents, then $s_1 = s_2$, and we may arbitrarily choose $p' = s_1$.

If both L_1 and L_2 are reordered (reordered(p, s_1) and reordered(p, s_2)) we terminate with a conflict. The algorithm stops when we arrive at $p' = n$. The rules for obtaining the next edit operation are summarized in table 1.

For each position p and next position p' we output a merged operation to L_m that reflects the changes in L_1 and L_2. The various combinations of operations in L_1, L_2, and the merged operation is shown in table 2. The merged operation in the cases when merging was less than obvious was determined by inspection of the use cases. One example is the handling of an insert wedged between two deletes (rows 3 and 4 in the table).

The algorithm thusly goes along the list of operations, outputting *regions that have not been deleted*, in the order that represents a change with respect to the original document order. Inserted text is always outputted, unless wedged between two deleted regions. In short, we follow the line of change.

To output the textual representation of D_m we loop over L_m, outputting the text of the region for the movi() operations, and any inserted text for the movi() and deli() operations. Figure 4 shows the merged script for the scripts in figure 3, as well as its textual representation. Comparing to the hand-merged result in figure 1, we see that our merge algorithm does indeed produce the desired result.

5.3.1 Handling Whitespace

To bring our merging algorithm closer to processing the modelled data, rather than its encoding, we introduce a step for *whitespace normalization*, where blocks of whitespace are replaced with a single space. This helps us produce more accurate matchings, and not introduce edits that are meaningless in the context of the modelled data.

We do, however, recognize that the formatting of the encoded data matters (indentation, linebreaks), and it should not be touched unless explicitly required. We therefore also provide the ability to map spaces back to their original blocks of whitespace. This is called *whitespace denormalization*.

5.4 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. This means, on one hand, 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 quiescently resolves ambiguous situations.

The only conflict that our algorithm recognizes is conflicting reordering of blocks, as listed in table 1. In addition, we output a conflict warning when co-located inserts are merged, as represented by rows 1 and 2 in table 2 when both t_1 and t_2 are non-empty, as well as in the case of inserted text in the middle of a deleted region (rows 3–4 in the table). Inspection of the study cases indicated that in most situations this will be a reasonable course of action.

6 Evaluation

We verified the extraction of requirements from the study cases by applying our merge to the cases. All of the 97 cases were successful. We also tested the tool on the 19 3dm "small" cases which did not include copy operations. Of these, 18 were successful, and 1 failed due to bad matching. We also did some preliminary testing on the full 3dm use cases, which entailed a total of 8 merging tasks. Of these, 3 failed due to bad matchings.

The tests show that more effort is required to tune the heuristics of the matching algorithm, whereas the merge phase works quite well. It appears that the full 3dm use cases were more self-similar than anticipated, which confused the matcher. We are, however, optimistic about finding better and more efficient matching heuristics.

We tested the scalability of the prototype by merging randomly edited XML documents of varying size. The results (figure 5) indicate that the algorithm scales satisfactorily with increased input size and a fixed number of edits.

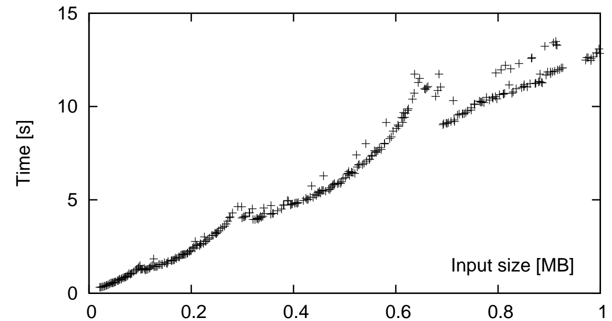

Figure 5. Merge times for 10 edits and varying input size.

7 Conclusions

The merge presented here represents initial research on constructing a text-based three-way merge for XML/HTML documents with support for document restructuring. A distinguishing feature of our merge is that the input documents need not be well-formed. Furthermore, by using a text- rather than tree-based approach we may perform edit detection using string matching rather than computationally expensive tree matching techniques.

Although our merge algorithm is very simple in its design, we have still obtained encouraging results on the applicability of the merge to its intended domain. The main weakness of the merge appears to be the heuristic matching phase, rather than the underlying theoretical model of processing inherently tree-structured data in a linear fashion.

References

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

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

[3] W3C, *HTML 4.01 Specification*, Dec. 1999. `http://www.w3.org/TR/html401/`.

[4] E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[5] J. MacDonald, "File system support for delta compression," Master's thesis, UC Berkeley, May 2000.

[6] M. K. Johnson, "Diff, patch, and friends," *Linux Journal*, vol. 1996, Aug. 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] T. Lindholm, "A three-way merge for XML documents," in *Symposium on Document Engineering (DocEng) 2004*, (Milwaukee, WI, USA), Oct. 2004.

[9] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, pp. 449–462, May 2002.

[10] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345–387, 1989.

[11] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 26–37, ACM Press, 1997.