Fuzzy matching for partial XML merge

Vladimir Fedotov
ISP RAS
Moscow
Email: vfl@ispras.ru

***Abstract*—In this paper we describe our experience in creating a tool capable to provide traceability of the requirements between different versions of the LSB and POSIX standards. We propose an approach for merging requirement markup without exact matching of the documents by means of arithmetic hashing and context similarity analysis.**

I. Introduction

In the past several years XML made huge progress, mostly due to its extensions. ISO finally standardized OpenXML and OpenDocument as document formats for office applications, thus making possible long standing dream of simple and robust document exchange in a heterogeneous environment.

At present day there are hundreds of XML extensions [1], applicable to a huge variety of tasks, from relational databases to vector graphics. Among the most important - XHTML providing transition from HTML to XML. Its importance relies on a huge amount of standards, specifications, recommendations and RFCs available in HTML and, with adoption of XHTML, available to processing by means of XML-aware tools.

Our present work is intended to provide requirement traceability between different versions of POSIX and LSB standards. Previous versions of these standards were converted to XHTML with requirements markup for the needs of OLVER [2] and other projects.

Actual markup consists of one or more span tags wrapping requirement-related text nodes. It is important to note here that span can actually divide text node to several chunks, therefore creating a sub-tree in place of one node, thus making impossible direct matching between marked and non-marked documents.

```
<parent>
  unrelated text
  <span requirement_id=1>
    The spice must flow!
  </span>
  unrelated text
</parent>
```

Listing 1. Markup example

So our goal may be defined as follows: given marked up previous version of the document, trace requirements chunks in the next version of the document and, if that trace was successful, merge requirement markup.

II. Problem description

Despite being straightforward, this problem actually represents a challenge to existing XML merging tools. Limited to open source Java implementations, we were unable to find a solution robust enough due to various reasons.

First of all, the given task requires zero faults merge, as a fault leads to document corruption. In case of XML merge, a fault is mismatching during the node mapping process, which can easily occur between nodes with the same names or values.

Second, it requires to trace a match between actually different nodes, as the markup can break up a text node in two or more chunks.

Third, the solution should take into account the context of matching nodes. For example, valid requirement for one document isn't valid for another document if it was moved to ”Deprecated requirements” section.

Finally, the solution has to be stable enough to work in the production environment. Unfortunately, most of the discovered tools were academic researches discontinued by their authors.

III. Related work

Tree diffing and merging remains a very active field of research, presently due to popularity of the bio-informatics. There is a variety of different techniques applicable to very different sets of problems. Very good review of the techniques available for XML was done by K. Komvoteas in [3].

Basically, an XML documents can be merged by direct comparison of their nodes, traversed in the definite order. Discovered conflicts fall into two categories: node insertion and node deletion. Some algorithms also recognize moving and updating, while others simply represent them as a consecutive deletion and insertion. Ruling out these conflicts is actually called a ”merge”.

Unfortunately, such direct comparison will often provide bogus results because nodes may only differ by their context. There are several approaches developed to compare node contexts.

DiffX algorithm [4] uses fragment matching technique. According to it, a node is considered a match if it has the largest matching neighborhood then any other equal node. Weakness of that approach is in corner cases – it is unable to provide any ruling in case of equal nodes with equal fragments, thus leaving us without guarantees of correct matching. Moreover, there is no simple way to perform a reverse task – to trace a single node by its context, in case we want to match actually different nodes.

A slightly different approach is proposed in [5]. So-called ”fingerprinting” is a way of storing node context in a vector of MD5 hashes. A node is considered a match if its fingerprint has the matching neighborhood within a given radius. As opposed to DiffX, it is possible to trace a node by matching its context, but such an approach will produce false results in case of matching several equal chunks with the same context, like brackets or commas enclosing reference links, which often occurs in documents.

Radically different is a three-way merge approach, which makes use of three documents instead of two. Often used in the revision control systems, three-way merge compares two target documents while also considering a ”base” document. In practice, it is implemented as a diff between the document α and the base, and then patching the document β with resulting delta.

In our case it could be implemented by considering the original unmarked document as a base, diffing it with the marked up document, and patching the target document with resulting delta. Diffing the unmarked base with the marked up document will produce the markup anchored to the actual nodes, which can be traced much more effectively in a target document. We are considering implementation of this approach in the future versions of our tool.

IV. Arithmetic hashing

To address the issues described above we propose an approach combining two different ways of evaluation of the node equality: by using positional metric and by comparing similarity of their text contents.

Each node has several attributes defining its position in the document: position in the traversal sequence, position among the siblings, depth, index among the identical siblings and overall siblings count. The problem is that these attributes should be properly weighted to be used in the positional metric and approach to provide such weighting isn’t clear. Which is more important for node equality: to have same depth or same traversal position? What if the node match candidate have exactly same siblings and depth, but is located in completely different place in document? And if another match candidate is located in exactly same place, but has no matching siblings?

In our implementation arithmetic hash provides us with cumulative positional metric. It uses general approach quite similar to arithmetic coding algorithm [6], but in completely different way.

Starting from the root, an interval is assigned to each node, with a random number from that interval assigned as its label. Each interval is divided by the number of node descendants and each descendant is provided with a range within parent interval according to its position among sibling nodes (fig. 1).

While being extremely simple this hashing technique provides us with positional metric which can be easily evaluated as a distance between the node labels (1). Its deliberately made biased towards matching of the nodes on the same axis, but being extremely sensitive to any changes in the siblings order providing better matching for equal text chunks, as opposed to [5].

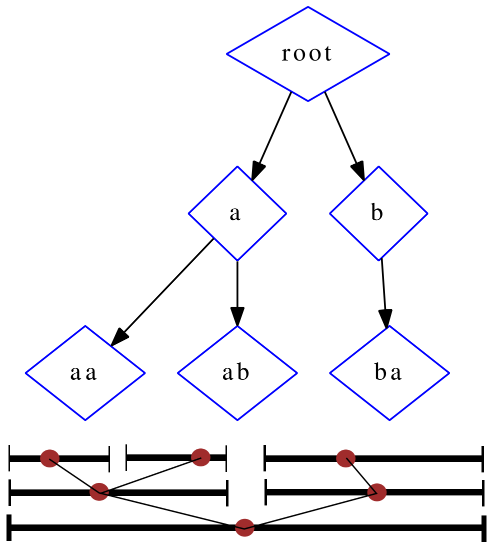

Fig. 1. Labelling example

$$distance(x,y) = \frac{|label(x) - label(y)|}{min(interval(x), interval(y))} \tag{1}$$

V. Context similarity

Bearing in mind that we are dealing with an XHTML documents, it is safe to say that almost any axis in such documents ends with a text node leaf. Therefore any ascendant nodes can be defined by the concatenation of the descendant text node values.

Evaluating similarity between text contexts provides us the similarity metric, that can be used for searching of the proper parent node, which later serves as a context for positional metric. Similarity itself can be evaluated by applying Jaccard index to the tokenized text contents (2).

$$jaccard(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

Being much faster than commonly used Levenshtein edit distance [7], Jaccard index provides satisfactory results, which can be further improved by taking into account the fact that requirement text should be equal or included into reference text. Therefore in our approach we use slightly modified similarity index (3).

$$similarity(A,B) = \frac{|A \cap B|}{A} \tag{3}$$

VI. Gluing it together

While none of the metrics described above can be used effectively separately, being combined they provide surprisingly robust results.

Similarity metric provides good results in case of comparison between nodes with diverse chunks of text, but fails if text contexts are exactly the same, which occurs quite often.

On the contrary, positional metric is ineffective in case of comparison between trees with strong differences, but becomes extremely effective for the small sub-trees.

Wrapping these metrics inside a stable sorting algorithm achieves the desired result. Firstly nodes are being sorted by their positional metric, then by their similarity metric. Therefore similarity metric has the priority over positional one, but in the corner case of several nodes having the same similarity index, they will be ruled out by their label distance, as opposed to [4].

VII. Conclusion

This paper presents a new approach to merge XML documents without exact matching of their nodes by using an algorithm combining node matching, based on the positional metric evaluated as label distance, with text similarity analysis based on the evaluation of modified Jaccard index between reference and target text nodes.

Despite being work in progress, our approach is already showing good results while merging different versions of LSB and POSIX standards as well as RFCs and many others.

In the nearest future we consider implementing three-way merge, based on our current approach and provide an open source Java implementation.

References

[1] http://en.wikipedia.org/wiki/List_of_XML_markup_languages
[2] http://linuxtesting.org/project/olver
[3] K. Komvoteas, *XML Diff and Patch Tool*, 2003.
[4] R. Al-Ekram, A. Adma and O. Baysal, *diffX: An Algorithm to Detect Changes in Multi-Verion XML Documents*, 2005.
[5] S. Ronnau, G. Philipp and U.M. Borghoff, *Efficient Change Control of XML Documents*, 2009.
[6] J.J. Rissanen, *Generalized Kraft Inequality and Arithmetic Coding*, 1976.
[7] V.I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.