Extreme Markup Languages 2004

Montréal, Québec
August 2-6, 2004

A Shallow Algorithm for Correcting Nesting Errors and Other Well-Formedness Violations in XML-like Input

Christian Siefkes
Berlin-Brandenburg Graduate School in Distributed Information Systems

Abstract

We argue that there are some special situations where it can be useful to repair well-formedness violations occurring in XML-like input, giving examples from our own work. We analyze the types of errors that can occur in XML-like input and present a shallow algorithm that fixes most of these errors, without requiring knowledge of a DTD or XML Schema.

A Conference of IDEAlliance
Extreme Markup 2004 Languages®

A Shallow Algorithm for Correcting Nesting Errors and Other Well-Formedness Violations in XML-like Input

Table of Contents

1 Introduction and Motivation....1
2 Types of Errors in XML-like Input....1
3 Configurable Settings and Heuristics for Repair....2
  3.1 Missing Root Element....2
  3.2 Widowed Start Tags....2
  3.3 Placement of Missing Start Tags....3
  3.4 Configuration of Character-level Errors....3
4 Algorithm Description....3
  4.1 First Pass....3
  4.2 Repairs at the Character Level....4
  4.3 Second Pass....4
  4.4 Serialization....5
5 Limitations....5
6 Further Application Scenarios....6
7 Related Work....6
8 Conclusion and Future Work....6
Footnotes....7
Acknowledgements....7
Bibliography....7
The Author....8

A Conference of IDEAlliance
Extreme Markup 2004 Languages®

A Shallow Algorithm for Correcting Nesting Errors and Other Well-Formedness Violations in XML-like Input

Christian Siefkes

§ 1 Introduction and Motivation

Well-formedness is an essential requirement that must be fulfilled by every XML document. There is a broad consensus not to accept any well-formedness violations and to reject any document that violates well-formedness instead of trying to fix the error.

However, there are some exceptions where it can be both necessary and safe to repair well-formedness violations, as long as this occurs not at the receiving side (XML parser) but at the generating side (XML generator). In this case the malformed document is never seen by other applications, it is only a temporary internal artifact of the generator.

A scenario from the field of Natural Language Processing (NLP) led to the development of the repair algorithm described in this paper. The goal is to prepare XML (e.g. XHTML) documents for text mining (information extraction) by augmenting them with linguistic annotations such as part-of-speech (POS) tags and sentence "chunks" (verb groups, noun phrases and prepositional phrases etc.). Such annotations can be conveniently stored in XML format. In case of plain text input there is no problem, but if the input already is XML the issue of nesting arises.

The tagger used in our project (*TreeTagger* [Tree]) is a third-party component mainly targeted at plain text input—in this case the output will be a well-formed XML document. It offers very limited support for XML input by ignoring any existing markup, simply copying it to the output. In this case, however, the resulting output will typically contain nesting errors and thus no longer be well-formed.

To address this and some other real-life problems (outlined in Section 6) we decided to develop an algorithm that can repair nesting errors and most other kinds of well-formedness violations in XML-like input. We denote the algorithm as "shallow" since it doesn't require access to a DTD or XML Schema to work.

In the next section we analyze the types of errors that can occur in XML-like input. We then explain the configuration options and heuristics used by our repair algorithm, prior to presenting the algorithm itself. After examining limitations of the algorithm, we outline further application scenarios and discuss related work. We conclude this paper by looking at possible further extensions of the algorithm.

§ 2 Types of Errors in XML-like Input

We distinguish several types of errors that can occur in XML-like input, preventing it from being well-formed.

1. **Character-level errors:** Errors at the character level, e.g. un-escaped "<" or "&" in textual content or unquoted attribute values.

```
Error                                  Possible Fix
<emphasis type=strong>                 <emphasis type="strong">
Procter & Gamble                         Procter &amp; Gamble
a < b                                    a &lt; b
</emphasis>                            </emphasis>
```

2. **Simple nesting errors:** Errors that can be fixed by swapping two tags.

```
Error                                  Possible Fix
<paragraph>                            <paragraph>
<sentence>                               <sentence>
...                                        ...
</paragraph>                             </sentence>
</sentence>                            </paragraph>
```

3. **Hard nesting errors:** Errors that can only be resolved by splitting an element.

```
Error                                   Possible Fix
<paragraph>                             <paragraph>
...                                       ...
<sentence>                                <sentence>
...                                         ...
                                          </sentence>
</paragraph>                            </paragraph>
<paragraph>                             <paragraph>
                                          <sentence>
...                                         ...
</sentence>                               </sentence>
</paragraph>                            </paragraph>
```

4. **Widowed tags:** "Widows" are singleton start or end tags whose corresponding end resp. start tag is missing.

```
Error                                   Possible Fix
<paragraph>                             <paragraph>
<sentence>                                <sentence>
...                                         ...
                                          </sentence>
</paragraph>                            </paragraph>
```

5. **Missing root element:** A missing root element affects the global structure of a document. We know that the root element is missing if there are several elements and/or textual content or CDATA sections at the outmost level of the document.

```
Error                                   Possible Fix
                                        <document>
<paragraph>                               <paragraph>
...                                         ...
</paragraph>                              </paragraph>
<paragraph>                               <paragraph>
...                                         ...
</paragraph>                              </paragraph>
Text.                                     Text.
                                        </document>
```

There are some other types of possible errors, e.g. concerning the uniqueness of attributes (duplicate attributes within a start or empty tag are prohibited) or the declaration of entities. These are not discussed here because they cannot be repaired by our algorithm (and most of them probably cannot be fixed in a generally useful way without user intervention).

§ 3 Configurable Settings and Heuristics for Repair

3.1 *Missing Root Element*

The last type of error (missing root) can only be fixed if the user specified the qualified name to use when a root element must be created (`document` in the example given above). If none is given and this type of error is detected, the algorithm gives up and declares the document as irreparable.

3.2 *Widowed Start Tags*

There are two options to process widowed start tags whose corresponding end tag is missing:

1. Either the missing end tag is created and inserted at a suitable position, e.g. immediately before the end tag of the embedding element.
2. Or the widowed tag is converted into an empty tag (this is equivalent to inserting a corresponding end tag immediately after the widowed tag).

```
Error                 First Option          Second Option
<paragraph>           <paragraph>           <paragraph>
<sentence>              <sentence>            <sentence/>
...                       ...                   ...
                        </sentence>
</paragraph>          </paragraph>          </paragraph>
```

To determine which option to use, a set of *emptiable* tags for which the second option should be used can be specified. The first option is used for widowed tags of all other types; in this case the missing end tag is inserted at the latest possible position (immediately before the embedding end tag).

3.3 *Placement of Missing Start Tags*

A simple heuristic for the placement of missing start tags is to place them immediately after the start tag of the embedding element (analogously to missing end tags). However if an element contains several widowed end tags of the same type (qualified name), the created start tags appear consecutively, resulting in a potentially deep nesting of same-type elements.

This might be appropriate in some cases, but more often same-type elements are arranged in succession within a common embedding element instead of being nested. Thus our heuristic is to place the first missing start tag of a type after the start tag of the embedding element, but to place any further start tags of the same type after the last end tag of this type.

```
Error                  Simple Heuristic       Our Heuristic
<paragraph>            <paragraph>            <paragraph>
                         <sentence>             <sentence>
                           <sentence>
...                          ...                    ...
</sentence>                </sentence>            </sentence>
                                                  <sentence>
...                          ...                    ...
</sentence>              </sentence>            </sentence>
</paragraph>           </paragraph>           </paragraph>
```

To realize this we use the concept of *tentative start tags*. A *tentative start tag* is created after an end tag whose start tag was either missing or itself *tentative* if the next tag of the same type is also an end tag (which means that another start tag is missing).

3.4 *Configuration of Character-level Errors*

For some kinds of *character-level errors* there are different possibilities of resolving them depending on user preferences. This is discussed in Section 4.2.

§ 4 Algorithm Description

Our algorithm proceeds in two passes. In the first pass, the input document is tokenized and *character-level errors* are fixed. All other kinds of errors are resolved in the second pass. The first pass also prepares suitable data structures to allow efficient repair in the second pass (data structures are marked by underlining in the following text).

4.1 *First Pass*

In the first pass, the XML-like input is tokenized into a sequence of constituents. XML documents can contain ten types of constituents:

1. XML declaration
2. Document type declaration
3. Processing instructions (PIs)
4. Start tags
5. End tags
6. Empty tags
7. Outer whitespace, i.e. whitespace preceding or following a tag or other markup
8. Textual content
9. CDATA sections
10. Comments

If there is text that doesn't fit any constituent type, the algorithm tries to fix this error at the character level, as described in Section 4.2. Tokenization is done via complex regular expressions, similar to the shallow XML parser described in [Cam98].

Constituents are either *markup* (declarations, PIs, tags, comments) or *text* (textual content, CDATA sections). Each markup constituent is assigned a *markup series number*—a *markup series* is a series of markup and outer whitespace not interrupted by non-whitespace text. The concept of *markup series* is used to distinguish between *simple* and *hard nesting errors*. Simple nesting errors can be resolved by moving tags within the same markup series.

In addition to a (doubly linked) list of all constituents, a data structure containing all unprocessed tags is built which initially contains all start and end tags (but no empty tags).

4.2 Repairs at the Character Level

These repairs are performed prior to the the detection of nesting errors to allow a correct tokenization.

1. **Escape illegal ampersands:** Any "&" characters occurring in textual content or attribute values (of start and empty tags) that do not start an *entity reference* or a decimal or hexadecimal *character reference* are escaped. The algorithm does not use a DTD, so it doesn't know whether or not entity references such as `&mdash;` are declared and thus legal. By default all possible entity references are accepted but the algorithm can also be configured to allow only character references and the five predefined entity references (`& < > &apos; &quot;`), while any other "&" characters are escaped even if starting a potential entity reference.
2. **Fix unquoted attribute values:** Attribute values whose start and/or end quotes are missing (`name=value`) or do not match each other (`name="value'`) are recognized and fixed (by enclosing them in full quotes and escaping any full quotes within the value). Unquoted values can contain any characters except "<", ">" and "=", they can even contain whitespace.
3. **Optionally delete "pseudo-tags":** We use the term "pseudo-tag" for character sequences that look similar to XML tags but are none. More formally, "pseudo-tags" start with a "<" character followed by any printable character, end with a ">" character, do not contain any embedded "<" or ">" and are not valid tags according to the XML 1.0 [XML1.0] or 1.1 [XML1.1] Specification. For example, `<0.05.12.91>` would be a "pseudo-tag". Optionally (off by default) "pseudo-tags" are deleted. Otherwise they are processed in the next step, i.e. the starting "<" character is escaped.
4. **Escape illegal characters:** Any remaining illegal characters (typically un-escaped "<" characters) are escaped.
5. **Optionally delete restricted control characters:** Optionally *restricted characters* (control characters in the ranges [#x1-#x8#xB-#xC#xE-#x1F]) are deleted. These characters are prohibited in XML 1.0 and discouraged in XML 1.1. This step is configurable and off by default.

4.3 Second Pass

A start tag is said to have a *corresponding end tag* if and only if the unprocessed tags data structure contains an end tag of the same type (qualified name), not preceded by a start tag of the same type.

A start tag is said to be *missing its end tag* iff the next unprocessed appearance is not an end tag and the number of unprocessed start tags of this type is equal to or greater than the number of unprocessed end tags of this type.

The second pass traverses the list of constituents created in the first pass. Each encountered start tag is moved from unprocessed tags to a stack of open tags. When an end tag is encountered, the algorithm iterates the following loop until the end tag has been processed (a match has been found):

1. **Check match:** If the end tag and the last open tag have the same qualified name, they match each other. The start tag is popped from open tags. When the matching start tag is a *tentative* tag and the next tag of this type is another end tag, we create a new *tentative* start tag of the same type and insert it after the matched end tag. Exit loop (done).
2. **Move tentative tag:** If the last open tag is *tentative*, it is moved after the current end tag (removing it from open tags and re-adding it to unprocessed tags). Go to step 1 (try to match preceding open tag).
3. **Find matching end tag:** If a *corresponding end tag* exists for the last open tag within the current *markup series*, it is moved before the current end tag. This is done only if a non-*tentative* start tag exists for the current end tag, otherwise we'll go to the next step (move or insert start tag for

current end tag) to avoid unnecessary tag movements. Start tag and matching end tag are popped from open tags resp. unprocessed tags. Go to step 1 (try to match preceding open tag).

This step fixes a *simple nesting error*.

4. **Find matching start tag:** If open tags contains a non-root tag matching the current end tag (either within the *markup series* of the last open tag or a *tentative* appearance anywhere), it is moved after the last open tag. The found start tag is popped from open tags. Exit loop (done).

   This step fixes a *widowed tag* (if the found tag is *tentative*) or a *simple nesting error* (otherwise).

5. **Insert missing start tag:** If open tags does not contain a start tag with the same type (qualified name) as the current end tag, we know that the start tag is missing and needs to be supplied. Thus a start tag of the same type (and without any attributes) is created and inserted after the last open tag.

   If the next appearance of this type is also an end tag, another start tag is missing—to provide it we create a *tentative* start tag and insert it after the processed end tag. Exit loop (done).

   This step fixes a *widowed tag*.

6. **Move premature start tag:** If the last open tag is within the current *markup series* and not *missing its end tag*, it is moved after the current end tag (moving it from open tags to unprocessed tags). Go to step 1.

   This step fixes a *simple nesting error*.

7. **Complete start tag:** If the last open tag is *missing its end tag*, it is convert into an empty tag (preserving any attributes) if it is *emptiable*; otherwise a matching (same-type) end tag is created and inserted before the current end tag. Pop start tag from open tags and go to step 1.

   This step fixes a *widowed tag*.

8. **Split element:** If none of the above conditions triggers, we know that the last open tag and the current end tag overlap. The only way to fix this is by splitting either of them in two parts. In the current implementation it is always the start tag (the last open tag) that is split. We split the last open tag by creating two new tags: (1) a matching (same-type) end tag that is inserted before the current end tag; (2) a copy of the start tag (including all attributes) that is inserted after the current end tag. Pop start tag from open tags and go to step 1.

   This step fixes a *hard nesting error*.

At the end of the document, end tags are created and added for remaining open tags, if any. They are inserted after the last *root content* (content that is only allowed within a single root element: tags, text except outer whitespace, CDATA sections), but before any trailing non-root content (outer whitespace, comments, PIs). This fixes *widowed tags*.

If the root element is missing, i.e. not all root content is enclosed within a single element and this cannot be fixed by moving tags within *markup series*, an root element of the configured type can be created. If the algorithm is not configured to create a root element (default), processing will stop with an exception in this case. The inserted root element will cover as little content as possible, i.e. all root content, but no preceding or following non-root content. This fixes a *missing root element*.

4.4 *Serialization*

After the two passes, any well-formedness violations that can be detected by our algorithm have been fixed. The repaired list of constituents is serialized into a document that in most cases will be well-formed XML (unless it contains errors that are not addressed by our algorithm, e.g. duplicate attributes).

§ 5 Limitations

While the heuristics of the algorithm are designed to cover typical problems in a reasonable way, there are some situations where the results will not be what a user might expect.

The heuristics for placing missing start or end tags cannot handle all cases adequately, especially they do not consider possible relationships between elements of different types. For example, in HTML [HTML4], the `th` and `td` elements are alternatives: a `th` element should end at the start of a `td` element, and vice versa. Since the algorithm is shallow and does not consider DTDs or Schemas, it cannot take such relationships into account.

In case of *hard nesting errors*, one of the two overlapping elements must be split, but there is no perfect way to decide which one. Currently the algorithm uses a very simple heuristic: it always splits the

element that starts and ends later (the second element). In some cases, a user might want to split the first element instead, but there is no way to detect this automatically.

Some combinations of errors can mislead the algorithm. If a *widowed start tag* is followed by a *widowed end tag* of the same type, the algorithm will assume that the end tag complements the start tag to form a single element. It will accordingly resolve any *hard nesting errors* between this presumed element and other elements, even if this means splitting an element multiple times.

Another kind of limitation results from the shallow treatment of attributes. When an element is split, any attributes are copied to the newly created start tag. In case of ID attributes this violates the ID validity constraint, since the ID value will no longer be unique. To complement a *widowed end tag*, a start tag without any attributes is created. This will cause a validity error if there are required attributes for this element type. These types of errors could only be addressed by accessing a DTD or XML Schema, if at all.

§ 6 Further Application Scenarios

In addition to the scenario presented in Section 1, we employ the algorithm in two other settings:

Sentence tagging: For our linguistic preprocessing, we need to insert elements enclosing whole sentences for augmenting the within-sentence level linguistic annotations provided by the tagger mentioned above. The tagger provides information that allows to locate the end of sentences, but it cannot detect the beginning. Thus we insert *widowed end tags* marking the end of sentences and let the algorithm insert the corresponding start tag based on the heuristic explained in Section 3.3.

Conversion of legacy documents: One of our text mining test cases is to extract information from the *RISE Seminar Announcements* corpus [RISE S.A.]. This corpus has been published in a format that is similar to but not exactly SGML (nor does it claim to be). This format uses start and end tags; but there is no root tag, characters such as "&" and "<" are not escaped, and the published documents contain lots of nesting errors (mainly of the *simple* kind). Our algorithm converts these documents into XML so they can be processed by any XML parser.

§ 7 Related Work

The shallow, regular expression-based *REX* [Cam98] parser has been an major source of inspiration for the tokenization performed in the first pass of the algorithm (though the regular expressions used here have been developed largely independently, partially due to the better Unicode support in Java and to address XML 1.1 [XML1.1]).

There are some programs that fix SGML/XML documents corresponding to certain DTDs. For example, *HTML Tidy* [Tidy] corrects errors in HTML documents, including nesting errors and missing end tags. Knowledge of used DTDs is built into such programs; they cannot be used for fixing documents conforming to other DTDs or XML Schemas.

There are algorithms for merging different versions of XML documents following a *diff and patch* model, e.g. [Kol03]. An overview of algorithms for detecting changes in XML documents in given in [Cob02]. The *3DM* system presented in [Lin01] performs a 3-way merge. Given the base form of a document and two variants created by independently editing the base form, a new version is created that unifies the changes performed in both variants. A similar approach is implemented in [Kom03].

For the problem at hand, such approaches would not be usable because they assume that (a) the different versions are correct XML and (b) all changes from the edited versions should be integrated. Thus it is not possible to impose new tree structure elements without being aware of the existing structure.

§ 8 Conclusion and Future Work

We have analyzed the types of errors that can occur in XML-like input, preventing it from being well-formed. We have presented a shallow algorithm that can fix most of these errors.[1] Repairing well-formedness violations is usually considered a no-no, but we have shown that there are situations where it can be useful, providing examples from our own work.

There are several paths for possible future work: the algorithm could be extended to fix other kinds of well-formedness violations, e.g. duplicate attributes; duplicate or misplaced XML/Document Type Declarations; or double hyphens ("--") within comments.

The heuristics used by the algorithm could be modified or refined. For example, missing end tags could be inserting adapting the placement heuristic that is currently used for missing start tags (cf. Section 3.3). A more challenging task would be to extend the algorithm to consider the DTD or Schema used for a document (if any exists), to address some of the limitations discussed in Section 5.

Notes

1. Our algorithm is freely available as part of the *TiEs* system [TiEs].

Acknowledgements

I would like to thank the anonymous reviewers for their useful comments and suggestions. This research is supported by the German Research Society (DFG grant no. GRK 316).

Bibliography

[Cam98] Cameron, Robert D. *REX: XML Shallow Parsing with Regular Expressions.* Tech.Rep. 1998-17, School of Computing Science, Simon Fraser University, 1998. **http://www.cs.sfu.ca/~cameron/REX.html**.

[Cob02] Cobéna, Grégory; Talel Abdessalem and Yassine Hinnach. *A Comparative Study for XML Change Detection.* Gemo Report 221, INRIA, 2002. **ftp://ftp.inria.fr/INRIA/Projects/gemo/gemo/GemoReport-221.pdf**.

[HTML4] *HTML 4.01 Specification.* W3C Recommendation, 24 December 1999. **http://www.w3.org/TR/html4/**.

[Kol03] Kohlhase, Michael and Romeo Anghelache. Towards Collaborative Content Management and Version Control for Structured Mathematical Knowledge. In *Second International Conference on Mathematical Knowledge Management (MKM 2003).* 2003. **http://link.springer.de/link/service/series/0558/bibs/2594/25940147.htm**.

[Kom03] Komvoteas, Kyriakos. *XML Diff and Patch Tool.* Master's thesis, Computer Science Department, Heriot-Watt University, Edinburgh, Scotland, 2003. **http://treepatch.sourceforge.net/report.pdf**.

[Lin01] Lindholm, Tancred. *A 3-way Merging Algorithm for Synchronizing Ordered Trees—The 3DM Merging and Differencing Tool for XML.* Master's thesis, Helsinki University of Technology, Dept. of Computer Science, 2001. **http://www.cs.hut.fi/~ctl/3dm/thesis.pdf**.

[RISE S.A.] RISE Seminar Announcements Corpus. **http://www-2.cs.cmu.edu/~dayne/SeminarAnnouncements/__Source__.html**.

[Tidy] HTML Tidy. **http://tidy.sourceforge.net/**.

[TiEs] Trainable Incremental Extraction System. **http://www.inf.fu-berlin.de/inst/ag-db/software/ties/**.

[Tree] TreeTagger. **http://www.ims.uni-stuttgart.de/projekte/corplex/TreeTagger/**.

[XML1.0] *Extensible Markup Language (XML) 1.0 (Third Edition).* W3C Recommendation, 04 February 2004. **http://www.w3.org/TR/REC-xml/**.

[XML1.1] *Extensible Markup Language (XML) 1.1.* W3C Recommendation, 04 February 2004, edited in place 15 April 2004. **http://www.w3.org/TR/xml11/**.

The Author

Christian Siefkes
Berlin-Brandenburg Graduate School in Distributed Information Systems, Database and Information Systems Group, Freie Universität Berlin
Berlin
Germany
christian@siefkes.net

Christian Siefkes is a Ph.D. student in Computer Science and a member of the Berlin-Brandenburg Graduate School in Distributed Information Systems. He is preparing his Ph.D. thesis in the area of information extraction, focusing on incrementally trainable statistical approaches. Former work included R&D on trust management in peer-to-peer architectures and the development and maintenance of a VoiceXML gateway for voice applications.

Extreme Markup Languages 2004
Montréal, Québec, August 2-6, 2004
This paper was formatted from XML source via XSL by Mulberry Technologies, Inc.