13 Merging Conflicting and Incomplete XML Markup

13.1 Introduction and Motivation

As explained in the preceding chapter, in the preprocessing stage documents are augmented 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. XML documents can only contain a single tree structure of elements—each elements must fit completely into an embedding element (its parent), overlapping elements are not allowed. However, since we want to use both the physical markup information of a document[1] and its linguistic structure, we have *two* independent (to a degree) tree structures for the same documents, and nesting conflicts between elements of the two structures are certain to occur at least occasionally.

Tools for linguistic preprocessing are generally targeted at plain text input; they will either prohibit or ignore document markup. The *TreeTagger* we use as POS tagger and shallow parser for our own system (cf. Sec. 12.1) generally expects plain text as input which will be converted into a well-formed XML document annotated with linguistic markup. 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.

Neither TreeTagger nor, to our knowledge, any other shallow parsers are able to correctly interweave the linguistic structure recognized by parsing with pre-existing document markup—indeed, this is a non-trivial problem with will be unsolvable if there are conflicts between the markup structure and the linguistic structure, unless suitable conflict resolution strategies have been defined. The repair algorithm described in this chapter has been developed with the goal of resolving such conflicts and merging such potentially conflicting tree structures into a single tree.

As mentioned in the previous chapter, the algorithm is also used for completing the sentence splitting process initialized by TreeTagger, by complementing the end-of-sentence markers (end tags) inserted by TreeTagger with corresponding begin-of-sentence markers (start tags).

To address these related problems we decided to develop an algorithm that can repair nesting errors and most other kinds of well-formedness violations in XML-like

[1] Which is explicitly given if input documents are in XML or (X)HTML format or else added to plain text documents by a heuristic recognition process (*txt2html*, cf. Sec. 12.1).

input. We use the term *XML-like input* to denote a document that is *meant* to be XML, even if it is not (due to well-formedness violations).

Previous approaches to information extraction have not used more than a single source of structured information, utilizing either linguistic information (most approaches aimed at "free-text" extraction) or markup information (*wrapper induction* approaches, cf. Sec. 5.3), but not both. This is probably the reason why this problem of merging structure information from potentially conflicting sources has not been addressed before (to our knowledge).

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.

13.2 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.

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>
```

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

```
ERROR                                   POSSIBLE FIX
<paragraph>                             <paragraph>
<sentence>                                <sentence>
...                                         ...
</paragraph>                              </sentence>
</sentence>                             </paragraph>
```

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>
```

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

```
ERROR                                   POSSIBLE FIX
<paragraph>                             <paragraph>
<sentence>                                <sentence>
...                                         ...
                                          </sentence>
</paragraph>                            </paragraph>
```

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. Such errors are not addressed by our algorithm for two reasons: first, they are not relevant for our own work since they do not occur in the preprocessed documents we need to handle; and second, they require user intervention to be resolved in a generally useful way (only a human user can decide which of the values of a duplicate attribute is the correct or most important one; or whether an unknown entity reference is a misspelling of some other entity or else whether and how it should be defined).

13.3 Configurable Settings and Heuristics for Repair

13.3.1 Missing Root Element

The last type of error (missing root) can only be fixed if a user specified a 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.

13.3.2 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). By default, we do not use any *emptiable* tags.

13.3.3 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 heuristic, we use the concept of *entative 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 indicates that another start tag is missing).

13.3.4 Configuration of Character-level Errors

For some kinds of *character-level errors* there are different possibilities of resolving them, depending on user preferences. They will be treated in Section 13.4.2.

13.4 Algorithm Description

The goal of the algorithm is to modify a document just as much as necessary to turn it into a well-formed XML document, but not more. Hence, all changes should be as non-intrusive as possible. This is the general design principle that underlies all the steps of the algorithm and motives the order in which they are executed.

To reach this goal, the 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 SMALL CAPITALS in the following text).

13.4.1 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 does not fit any constituent type, the algorithm tries to fix this error at the character level, as described in Section 13.4.2. 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).

13.4.2 Repairs at the Character Level

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

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 does not 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 (`&amp; &lt; &gt; &apos; &quot;`), while any other "&" characters are escaped even if starting a potential entity reference.

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.

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 [XMLa] or 1.1 [XMLb] 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.

Escape illegal characters: Any remaining illegal characters (typically un-escaped "<" characters) are escaped.

Optionally delete restricted control characters: Optionally *restricted characters* (control characters in the ranges [0x1–0x8], [0xB–0xC], [0xE–0x1F]) are deleted. These characters are prohibited in XML 1.0 and discouraged in XML 1.1. This step is configurable and off by default.

13.4.3 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 will 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 and 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*, a 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*.

13.4.4 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).

13.5 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 [HTM], 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 does not consider DTDs or Schemas, it cannot take such relationships into account. Requiring a DTD or Schema would conflict with the purposes of our information extraction system, since we want to be able to process any XML-based text files regardless of the exact format (cf. the input requirements defined in

Section 7.2.1), so the documents to process might not correspond to a DTD or the corresponding DTD might be unknown.

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.

13.6 Application in Our Approach

In addition to unifying the original structural markup with the linguistic annotations added during preprocessing into a joint tree structure (cf. Sec. 13.1), we employ the algorithm for two other purposes:

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. As described above, the tagger provides information that allows locating 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 13.3.3.

Conversion of legacy documents: One of our IE test corpora is the *RISE Seminar Announcements* corpus ([RISb], cf. Sec. 17.1). 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 to inline-annotate answer keys (cf. Sec. 9.3.1); 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 such documents into XML so they can be processed by any XML parser, allowing correct recognition of the annotated answer keys.[2]

[2] Meanwhile, an XML version of the *Seminar Announcements* has been published by the University of Sheffield’s *Dot.Kom Project* <`http://nlp.shef.ac.uk/dot.kom/resources.html`>, but it was not yet available when we started our experiments on that corpus.

13.7 Related Work

Since the problem of merging conflicting and incomplete XML markup is quite distinct from the issues normally covered in the field of IE, we discuss the related work in the context of this chapter (instead of in Part I, which is dedicated to related IE approaches and the field of information extraction in general).

The shallow, regular expression–based *REX* [Cam98] parser has been a 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 [XMLb]).

There are some programs that fix SGML/XML documents corresponding to certain DTDs. For example, *HTML Tidy* [Tid] 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. [Koh03]. 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.