Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 243–250
ISBN 978-83-60810-22-4
ISSN 1896-7094

LingURed: Language-Aware Editing Functions Based on NLP Resources

Cerstin Mahlow and Michael Piotrowski

University of Zurich
Institute of Computational Linguistics
Binzmühlestrasse 14, 8050 Zürich, Switzerland
{mahlow, mxp}@cl.uzh.ch

***Abstract*—In this paper we compare the state of the art of language awareness in source code editors and word processors. *Language awareness* refers to functions operating on the elements and structures of a formal or natural language. Language-aware functions allow users to work with meaningful units, increasing efficiency and reducing errors. While all modern source code editors provide programmers with language-aware functions, similar functions for natural-language editing are almost nonexistent. Writers have to manipulate characters, which makes editing and revising challenging and results in typical errors. We describe the LingURed project, in which we implement language-aware editing functions for German with the goal of supporting experienced writers. Our approach is based on the combination of standard editor functionality and shallow localized natural language processing. Prototypical functions demonstrate the feasibility of the approach. Based on our preliminary experience we discuss requirements for NLP components suitable for use in interactive editing environments.**

I. Introduction

IN THE context of text editing, *language awareness* refers to functions operating on elements of a certain language and respecting its rules. Currently, language awareness is almost exclusively found in source code editors, so the languages in question are programming or markup languages. Editors offering such functions are called *language-aware editors*. Language-aware editing systems exploit the knowledge of the language of a text to "provide services beyond the scope of traditional text editors" [1, p. 431]. The goals of these services are to increase convenience and efficiency, and, most importantly, to prevent syntactic (and some semantic) errors as early as possible. A central feature are editing functions operating on the structural elements of a certain language.

In general, we can distinguish *information functions*, *movement functions* and *operations*. Information functions are used for highlighting structural elements or for gathering information regarding certain elements and structures; they don't change the code or the text. Movement functions help the user to easily set the cursor on a certain position: For example, a programmer can jump to the next keyword or to the end of the previous loop. Operations actually change the text. They are used for inserting, reordering, modifying, or deleting elements, e.g., conditional expressions, functions, or blocks. The operation of all three types of functions is governed by the lexicon and the syntactic rules of the language.

While language awareness is currently only found in source code editors, the same principles could also be applied to natural-language text: Just as program text, natural-language text is not merely a sequence of characters and lines, but has an internal structure made up of words, phrases, clauses, sentences, paragraphs, and other elements. Obviously, natural languages have lexicons, and morphologic and syntactic rules. In fact, programming languages are, for the most part, inspired by natural languages, so the similarity is no coincidence. The objectives of language awareness in source code editors—convenience, efficiency, and error prevention—would also be desirable for natural-language text editing.

In section II we will briefly cover aspects of language awareness in source code editors and word processors. In section III we will show some of the challenges for writers editing and revising text with a word processor. In section IV we will outline the concept for the LingURed project to offer writers support to overcome these challenges. In section V we will describe aspects of our prototypical implementation, with emphasis on the inclusion of computational linguistics resources and with a side note on requirements for NLP components for real-world applications in section VI.

The LingURed project aims to implement language-aware functions for German to support experienced writers. Therefore most of the examples in this paper are in German (with translations in footnotes where needed). German is interesting for our research since it poses some complex linguistic challenges that cannot be solved using simple pattern matching methods, and which are not present in English.

II. Language awareness in editors

Although there are many parallels between natural and artificial languages and although language awareness—as a concept—can be adapted to source code editors intended for writing computer programs as well as to word processors intended for writing natural-language text, there are important differences.Consequently, the state of the art for language-aware editing differs between source code editors and word processors.

```
/*
 * Print unknown arguments.  TODO: Print usage.
 */

if (argc != 1)
{
   int i;
   fprintf(stderr, "%s: Unknown arguments:", argv[0]);
   for (i = 1; i < argc; i++)
   {
      fprintf(stderr, " %s", argv[i]);
   }
   fputc('\n', stderr);
}
```

Figure 1. Syntax highlighting and indentation of source code

A. Language awareness in source code editors

Language awareness in source code editors benefits from the relatively small and closed lexicon and strict syntactic rules defined *a priori*. Therefore even sophisticated processing of programming language source code can generally be implemented reliably and with relative ease.

The most striking evidence of language awareness in source code editors are indentation and syntax highlighting[1]. Based on the lexicon of a programming language (i.e., the keywords and the rules for names) and its syntactic rules, elements of the source code are highlighted and lines are indented according to structural and syntactic criteria. Both features are meant to give programmers a good overview of the code and to enable them to see the structure at a glance—indentation usually helps in identifying the body of loops, function definitions, etc. For programming languages where indentation has a syntactic function, e.g., Python, the editor assists in maintaining the syntactically correct indentation. Figure 1 shows a portion of a program as displayed in the XEmacs editor: Various elements of the code (e.g., comments, type identifiers, keywords, and strings) are highlighted in different colors; the editor also automatically maintains the correct indentation.

Another aspect of language awareness is the support the programmer gets during coding: The editor balances bracketing structures, it may fill in default parameters or ask the programmer to set the relevant values, thus helping to not forget to indicate a mandatory argument. The editor can insert or complete syntactic structures, e.g., conditional expressions or looping constructs. Further support depends on the language a programmer is using and may involve static semantic analysis to help detect semantic errors.

B. Language awareness in word processors

Natural languages have a large lexicon which can be extended *ad infinitum* by applying the morphologic rules of the language. Syntactic rules are not formally defined (as for programming languages) but only described *a posteriori* by linguists. More importantly: The language users change the rules and the lexicon as they are using their language—natural languages "live." Therefore processing natural languages is difficult and needs more resources in terms of time and space.

The checkers for spelling, grammar, and style offered by many word processors should not be confused with language awareness. Checkers are effectively post-writing tools: They do not support writers *during* composition, but offer critique on finished text. There is no interactive support for writers as we can find it for programmers. In fact, Daiute and Taylor define an editor for writing natural language texts as "a computer program with the capabilities for editorial assistance—the capabilities to manipulate strings of letters, numbers and other characters" [3, p. 84], i.e., there is no mention of words or sentences.

Today's word processors, for the most part, still offer only functions that operate on characters and lines, although there have been several attempts to create language-aware authoring aids in the 1980s and 1990s (see [4], [5], [6], [7]). However, none of these systems were developed beyond research prototypes.

III. Challenges while writing, revising and editing using word processors

As today's word processors offer no functions operating on language units, writers have to concentrate on the "brain-to-hand-to-keyboard-to-screen connection" [8, p. 79]. Various studies show that revising and editing are very demanding in terms of cognitive load (see [9], [10], [11], [12]). Writers have to translate their editing intentions—which they could often describe easily using linguistic terms (e.g., *Merge these sentences* or *Rewrite without modal verbs*)—into complex sequences of character-level commands, execute these commands, and verify the results. This is tedious and error-prone and leads to typical errors in texts, such as missing finite verbs, duplicate verbs, agreement errors, or wrong word order. Many of these errors can be reconstructed and explained as *slips* (according to the classification of Norman [13]), in particular *misordering the components of an action sequence*, *capture errors*, and *forgetting an intention*.

For example, changing a main clause into a subordinate clause (which is necessary when sentences are combined, or when it is intended to modify a statement) the writer of a German text has to be aware that the verb-second word order in main clauses has to be changed to verb-final word order in subclauses. Conversely, when splitting a long sentence into two (or more) shorter ones, it can be necessary to change subordinate clauses into main clauses or to complete shortened coordinate clauses.

This type of revisions is frequently the cause for duplicate or missing finite verbs, as can be seen in this example from an actual text:

(1) Zusätzlich ist zu berücksichtigen, dass das Werkzeug **hat** natürlich auch Einfluss auf den Prozess **hat**.[2]

Probably the original version of this sentence was:

[1]The term *syntax highlighting* is an "unfortunate misnomer" [2], since in most cases lexical, not syntactic, elements are highlighted.

[2]Free translation: 'In addition, one has to consider that the tool obviously influences the process.' The finite verb is set in bold.

(2) Das Werkzeug **hat** natürlich auch Einfluss auf den Prozess.[3]

The writer correctly added the finite verb at the end of the subordinate clause, but forgot to remove the verb in the original position.

In long German sentences with several coordinated main clauses sharing the same finite verb, it is possible to only use the verb in the first clause and omit it in the following ones. When splitting such a sentence into several independent main clauses, the writer must, of course, reinsert the verb. Example 3 shows one long sentence (finite verbs in bold); example 4 shows the resulting two sentences after the author had split the original sentence (note that the second sentence misses the finite verb); example 5 shows the intended correct version.

(3) **Ist** eine erste Fassung erstellt, **werden** häufig Meinungen von Lektoren und Kollegen eingeholt und deren Änderungsvorschläge in einer zweiten, dritten, vierten Fassung in den eigenen Text eingearbeitet.[4]

(4) **Ist** eine erste Fassung erstellt, **werden** häufig Meinungen von Lektoren und Kollegen eingeholt. Anschliessend deren Änderungsvorschläge in einer zweiten, dritten, vierten Fassung in den eigenen Text eingearbeitet.

(5) **Ist** eine erste Fassung erstellt, **werden** häufig Meinungen von Lektoren und Kollegen eingeholt. Anschliessend **werden** deren Änderungsvorschläge in einer zweiten, dritten, vierten Fassung in den eigenen Text eingearbeitet.

The situation gets even more complicated when splitting sentences consisting of several coordinated subclauses and main clauses.[5]

As Piolat points out, when revising and editing "writers must successively make a series of corrections, while checking to see that each one is compatible with others, often located at different linguistic levels" [10, p. 266]. It is hard work to achieve the original intention towards a communicative goal and at the same time take care of linguistic aspects of the text—and to execute the necessary word processor commands.

Writing is a creative task. Writers often "play" with language to find the best way of saying something. For example, a conjunction like

(6) objects and structures of natural languages

has a number of possible variations, such as:

(7) structures and objects of natural languages

(8) objects of natural languages and structures

The examples 7 and 8 are similar in meaning to example 6, but each variant has a different focus and, depending on the context, may better express the communicative goal of the author. To choose the best option, the author often needs to *see* the variants in context, try another option, undo a decision, etc. However, swapping conjuncts requires several steps including cursor movements and marking, copying, and pasting stretches of characters. Playing with different formulations is not possible, as very soon the phrase will be completely messed up. Wouldn't it be nice not having to deal with characters, but to have commands operating on a higher level, so that one could concentrate on the text, try different variations, and then choose the most appropriate one? Writers with powerful commands at their disposal would be relieved from the task of translating their high-level rhetoric or stylistic goals into long, complex sequences of low-level character-based operations.

IV. The LingURed project

In the LingURed project[6] we are developing language-aware functions to support writers during revising and editing. We are working for German; the target group are experienced writers. Their experience can be located on several levels: They are native speakers of German and have an extensive repertoire of stylistic devices for achieving their communicative intentions, they write on a professional level, and they are used to write with word processors.[7] These users are usually not willing to change their editing environment (see [1], [2]), but they would welcome additional support. Therefore we are not developing a new editor, but extending an existing one, namely XEmacs[8], a variant of Emacs [16] (see section V for a more detailed rationale for this decision).

Above, we have proposed language-aware editing functions for writing natural language text by referring to similar functions available for programming languages, arguing that, just like programming languages, natural languages can be described as a set of a lexicon and morphologic, syntactic, and semantic rules. When looking at the degree of language processing used in language-aware source code editors, we face similar questions as for natural language processing: While being edited, text is most of the time ill-formed, incomplete or inconsistent. Processors or parsers in interactive environments have to deal with these characteristics (see [17]).

When thinking of natural language processing, one may imagine deep, robust, and complete syntactical and semantical analysis. In fact, however, the results of today's parsers for German are not as satisfying and reliable as they would need

[3]Free translation: 'The tool obviously influences the process.' The finite verb is set in bold.

[4]Free translation: 'When the first draft is finished, usually opinions from colleagues and editors are requested and will be considered for a second, third, fourth draft.'

[5]Note the movement of the verb when changing *Es **ist** üblich, dass Meinungen [...] eingeholt und deren Änderungsvorschläge [...] in den eigenen Text eingearbeitet **werden***. into *Es **ist** üblich, dass Meinungen [...] eingeholt **werden**. Anschliessend **werden** deren Änderungsvorschläge [...] in den eigenen Text eingearbeitet.*

[6]LingURed stands for "Linguistically Supported Revising and Editing"; see also `http://www.lingured.info/`.

[7]These writers share the three kinds of awareness and the four skills described for *professional writers* in [14]. They differ from novice writers with respect to the knowledge of the topic of the text, the linguistic knowledge, and the strategies at their disposal to deal with all aspects of the writing process [15].

[8]The XEmacs Web site is at `http://xemacs.org/`.

to be for real-world applications. The question is thus: Is complete parsing really necessary for implementing language-aware editing functions?

Obviously, if sophisticated processing is possible, these sophisticated analyses can provide better resources for more sophisticated services. For programming languages, it is no problem to create a full, correct parse tree for a correct program. However, to ensure that the program is correct at all times, the user would have to be restricted to correctness-preserving operations on the parse tree without invalid intermediate states. Experience with so-called *syntax-directed editing environments* [18], [19] has shown that this is an unacceptable restriction for users.

Van De Vanter and Boshernitsan thus argue that for language-aware source code editors, "the amount of language analysis performed [should be] as simple (and localized) as possible, but also as useful as possible" [2, p. 4]; they conclude that the optimal amount lies somewhere in between using lexical tokens on one side and parse trees on the other side.

Given the current state of the art of natural-language parsers, it would be completely unacceptable (outside of specialized applications) to restrict authors to only those sentences the system can parse. Generally, in NLP one can say that the deeper the analysis, the lower the quality and reliability. This means that deep analysis can only theoretically provide interesting services, as it is currently not possible to reliably analyze unrestricted text.

Adapting the design principles from [2] to natural language, we can place the maximum degree of language processing needed for implementing useful editing support between morphological analysis and shallow syntactical parsing. Most of the proposed functions require only relatively low-level analyses, e.g., POS tagging (see section V).

Thus, one principle of our approach is to keep analyses as simple and localized as possible in order to minimize the resource requirements and to maximize robustness; it is not necessary to parse the *entire* text written so far (and to work around incomplete sentences) to highlight the subject or the finite verb in the sentences the writer currently sees on the screen. At the same time, we are trying to make the functions as useful as possible with the resources available.

A second principle is related to the selection of functions to implement. Here we can also rely on experiences from the development of source code editors.

One goal for the development of the EVE editor for the VAX was to reduce the cognitive load of the users. Based on logging data, the developers decided "to include commands that were used moderately frequently [. . .], as well as powerful commands that would be difficult or impossible to build out of a series of more elementary commands, even if these powerful commands are not so frequently used" [20, p. 95]. We are using this as a guideline for the implementation of functions in the LingURed project.

V. Prototype implementation

We have started to implement language-aware editing functions for the XEmacs editor. Due to its architecture, XEmacs is ideally suited as a test bed for new concepts: almost all of its functionality is implemented in Emacs Lisp and can thus not only be inspected, but also adapted and replaced at run-time. Furthermore, all Emacs Lisp functions are first-class citizens and have full access to all editor objects, i.e., there is no difference between functions that happen to be shipped with the XEmacs distribution and functions you write yourself or functions from third-party packages. This is in sharp contrast to the architecture of, for example, Microsoft Word: It is not possible to change the standard behavior of Word, one may only add functionality through so-called *add-ins*, which only have access to those objects and data structures the vendor has selected to expose via the API.[9]

At this point, our main goal is not completeness; we rather want to explore several types of functions and evaluate the usability of both the functions and the linguistic resources.

Depending on the task, language-aware functions require more or less linguistic knowledge, analysis, and resources (see [21]). In section IV, we argued for using as simple language analysis as possible. The functions are thus implemented by combining functionality already available in XEmacs, new Emacs Lisp functions, and external linguistic resources and tools as required.

A. Linguistic resources

To make POS information available to Emacs Lisp functions, we have implemented an XEmacs interface to the Mbt part-of-speech tagger [22]. This interface currently provides one function, `postagger-tag-sentence`, which sends the current sentence to the tagger and attaches the POS tags returned by the tagger as so-called *text properties* to the word forms of the sentence. Text properties are a mechanism for attaching "invisible" information to characters in an XEmacs buffer. This interface provides basic functionality for accessing linguistic information on which higher-level functions can be built.

For morphological analysis and generation we are using the GERTWOL morphological analyzer/generator [23], which we have integrated into XEmacs in a similar way as the Mbt tagger.

We will now describe three functions and their implementation to illustrate different types of functions and the range of functionality we are considering in the LingURed project. We will then give an outline of functions we consider for implementation on the basis of the functions described above. In section VI, we discuss requirements that NLP resources must meet to be suitable for use in interactive editing applications.

[9]Three of our students (Janick Bernet, Sandro Coretti, and Christian Oberholzer) have successfully implemented some functions in a Microsoft Word add-in. However, add-in development is complex and hampered by a poorly documented and deficient API, making Word unsuited for experimentation.

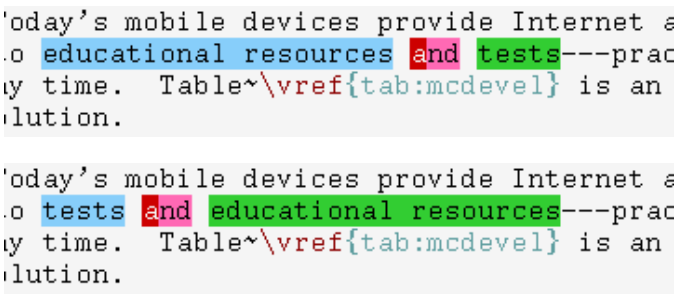

Figure 2. Transposing conjuncts

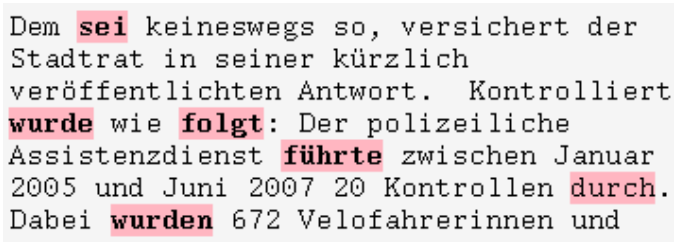

Figure 3. Highlighting Verbs

B. *Transposing conjuncts*

One of the functions we have already implemented provides support for swapping the elements of conjunctive constructions. It allows authors to easily change, for example, *teachers and students* into *students and teachers*. The elements of the conjunction (the *conjuncts*) are not limited to single words. This task actually seems to be more frequent than one may think, and it has some interesting properties: Even though the task is conceptually simple, it requires a complex sequence of commands in word processors. For example, the optimal keyboard command sequence for transposing two single-word conjuncts in Microsoft Word requires eight steps [24]; research also shows that writers rarely use the "optimal" command sequence, but typically more complex ones [25]. Authors are thus likely to make errors during this operation, and, especially if it is only executed rarely, they are prone to forget steps. Finally, such changes are often stylistically motivated: Stylistic decisions are typically made after comparing different variants and considering their effect in context. This is therefore a situation in which authors could especially profit if they could quickly "play" with their words without the risk of introducing errors (see section III).

Our XEmacs implementation works as follows: To swap two elements around a conjunction, the author places the cursor on the conjunction and invokes `conjunct-mode`; the conjunction and the words immediately to the left and right of the conjunction are highlighted (see figure 2, *top*). The normal editing keys are disabled in `conjunct-mode`; instead special key bindings are available to extend, move, or shrink the selection word-by-word for the left and right conjuncts. Once satisfied with the selection, the author presses t to transpose the conjuncts (see figure 2, *bottom*). Pressing t again reverts the transposition. The extent of the conjuncts can always be readjusted. Once satisfied with the result, the author exits `conjunct-mode` with the customary C-c C-c. As usual in XEmacs, all changes can be reverted using `undo`.

Note that extending or reducing the selection means that each keypress makes it one word longer or shorter, i.e., the selection mechanism is aware of the linguistic unit "word." At an earlier stage, we experimented with automatic detection and selection of potential left and right conjuncts, such as noun phrases or prepositional phrases. However, there are often multiple possibilities and second-guessing what the author may want to do is rarely successful. In effect, this approach would be similar to syntax-directed editing environments, which are known to be to restrictive (see also [26, p. 17]).

`conjunct-mode` is an example of a language-aware function which can be built on standard XEmacs functionality. It is language-aware since it supports an editing task specific to natural languages. Its level of language awareness is, of course, relatively low, nevertheless it encodes a number of facts about a class of languages (while it is usable for many languages, e.g., German, English, French, Italian, it does not support languages where conjunctions are realized as suffixes, such as *-que* in Latin).

C. *Syntax highlighting*

Many more advanced functions require part-of-speech (POS) information. A simple example of a function that makes use of POS information is the highlighting of certain parts of speech, e.g., finite verbs, nouns, or certain combinations of parts of speech.

While it is unlikely that authors have highlighting turned on at all times (as in programmer's editors), highlighting may help in some situations, especially with complex constructions. For example, German has a large number of verbs with *separable prefixes*, i.e., prefixes which are in some cases separated from the verb; the prefix is then placed at the very end of the sentence. It is thus easy to miss a separated prefix. Highlighting verbs may thus help to identify potential problems.

Figure 3 shows an excerpt of a text in which finite forms of full and auxiliary verbs are highlighted by shading and boldface type. Separated prefixes of verbs are shaded only (in this example the prefix *durch* of the verb *durchführen* 'perform', which here occurs in the form *führte* ... *durch*).

D. *Replacing words*

POS information is also necessary for operations that actually change the text globally, e.g., to replace all word forms of one word with the corresponding word forms of another word. This is a relatively simple task in English, but a challenge in highly inflecting languages, such as German, where words can have many word forms and each word form can typically express more than one category (see table I for the paradigms of two German nouns).

Manually replacing all occurrences of *Zelt* 'tent' with the corresponding word form of *Haus* 'house' is therefore a complex task: First, one has to find all word forms of *Zelt*—using

Table I
WORD FORMS OF *Zelt* AND *Haus*.

Word	Forms	Categories
Zelt (*n*, (e)s/e decl.)	Zelt	NomSg, DatSg, AccSg
	Zeltes	GenSg
	Zelts	GenSg
	Zelte	DatSg, NomPl, GenPl, AccPl
	Zelten	DatPl
Haus (*n*, (e)s/er decl.)	Haus	NomSg, DatSg, AccSg
	Hauses	GenSg
	Hause	DatSg
	Häuser	NomPl, GenPl, AccPl
	Häusern	DatPl

a conventional search function, this would require searching for each word form individually. Then, one would have to determine the category of a specific occurrence; note that the word form may be ambiguous, and the exact category can only be determined by looking at the syntactic context. Finally, one would have to manually replace the word form of *Zelt* with the corresponding word form of *Haus*.

We have therefore implemented the function `query-replace-word`, which works as follows: After calling the function, the writer is prompted to enter the word to replace (*from-word*) and its replacement (*to-word*). Each of the two words is analyzed morphologically; if a word is ambiguous (e.g., if it could be a verb or a noun), the writer is prompted for confirmation. POS information is used to present the contextually most likely choice as the default option. For a replacement to be possible, both *from-word* and *to-word* obviously must be of the same part of speech. Next, the paradigm of *from-word* is generated and the word forms are searched for in the text. Since different words (often with a different POS) may share some word forms, we check the POS information at each occurrence for compatibility. If the writer confirms the replacement, the word form of *from-word* at the current occurrence is analyzed to determine its category, and the corresponding word form of *to-word* is generated and the replacement is performed.

E. Further functions to be implemented

In section I, we distinguished information functions, movement functions, and operations. Information functions rely on linguistic analysis for morphological and syntactical information. Using tagging (as described in section V-C) we can highlight word forms belonging to a certain POS, e.g., prepositions, nouns, or adverbs. For example, highlighting conjunctions could help to get a better overview of the argumentative structure of the text or help to identify elements that may be overused [27]. By combining tagging and morphological analysis we can identify word forms according to more detailed criteria, e.g., finite verbs in the past tense or nouns in nominative singular. The user can also ask for statistical information, e.g., the sequence of verbs used in the text written so far[10] or the frequency of the conjunctions used in the text. Using the POS of the word forms in the text written so far we can also highlight sentences or phrases *missing* certain elements, e.g., sentences without finite verb.

The availability of part-of-speech information also makes it possible to implement language-aware movement functions. Movement functions are built upon information functions. Listing 1 shows a simple example of a function making use of POS information to position the cursor on the first finite verb of a sentence (`VVFIN` is the corresponding tag in the STTS tag set [28]).

Similarly, part-of-speech information can be used for selection functions, e.g., to select the phrase to which the word at the current cursor position belongs, or for operations like deleting the current phrase. Here we combine new functionality, i.e., selecting certain elements based on language-dependend criteria, with existing functionality, i.e., deleting or moving selected elements. These functions belong to the class of operations as they actually change the text. We can further distinguish global and local operations. An example for a global operations is the function `query-replace-word` described in section V-D. Another global operation would be to generally change the verb tense from present to past throughout a text. An example for a local operation is the function for transposing conjuncts described in section V-B. When calling a local operation, the writer actually sees the text passage the function is supposed to change. Examples are merging or splitting sentences, or changing a specific noun phrase from plural to singular.

Table II gives an overview of the functions we are considering for implementation.

VI. RESOURCES FROM COMPUTATIONAL LINGUISTICS AND THEIR QUALITY

LingURed is a real-world application making use of components and resources from computational linguistics. As outlined in section IV, we don't integrate sophisticated parsers, but only basic NLP tools, such as a tagger or a morphologic component for analyzing and generating word forms as described in sections V-C and V-D. To be usable in the LingURed framework, these components have to meet certain requirements: They must be (1) fast and (2) robust; the results should (3) be reliable and (4) should be returned in a format suitable for further processing; ideally, the components should be (5) open-source.

While one can find many references to linguistic components in the literature, much fewer systems are actually available; even fewer are suitable for embedding in an application and usable in an interactive environment. For example, the widely used TreeTagger [29] is not suitable for interactive use, since it uses buffered I/O and there is no reliable way to force the analysis of a short chunk of text; since TreeTagger is not open-source, it is not possible to change this behavior.

[10]For example, the sequence of verbs in the abstract of this paper is: "compare, refer, allow, work, increase, reduce, provide, be, manipulate, make, result, describe, implement, support, base, demonstrate, discuss."

```
(defun goto-first-finite-verb ()
  "This is an example function which shows how POS information stored
in text properties (as added by 'postagger-tag-sentence') could be
used."
  (interactive)
  (beginning-of-thing 'sentence)
  (let* ((start (car (bounds-of-thing-at-point 'sentence)))
         (end   (cdr (bounds-of-thing-at-point 'sentence)))
         (verb  (text-property-any start end 'pos 'VVFIN)))
    (when verb
      (goto-char verb))))
```

Listing 1. Example movement function making use of POS information

Table II
LANGUAGE-AWARE FUNCTIONS

Class	Function
Information	• Highlight elements or structures with respect to their linguistic category, i.e., word forms of a certain POS (e.g., verbs), or word forms belonging to a certain category (e.g., verbs in first person plural past tense), or phrases of a certain type (e.g., prepositional phrases). • Highlight sentences missing certain elements or structures • Report sequence and frequency of certain elements or structures • Report on variants of multiword terms and highlight them
Movement	• Move the cursor to the beginning or end of the next element or structure with certain characteristics • Move the cursor to the beginning or end of the previous element or structure with certain characteristics
Operations	• Change all elements or structures belonging to a certain class (POS or category) with respect to certain parameters • Replace words or phrases • Copy, paste, delete, or move elements or structures with certain characteristics • Change a single element or structure with respect to certain parameters • Merge two sentences • Split a sentence

Luckily, with Mbt there is an open-source tagger which was designed with integration in mind. Even though one may think of morphologic analysis as a solved problem, only very few systems are actually available for German. We found only GERTWOL to fulfill all requirements; unfortunately, it is not open-source, but there currently exists no usable open-source alternative for German. See [26] for a general discussion of limits of current linguistic components for interactive editing and [30] for an evaluation of currently available morphology components for German.

VII. SUMMARY AND OUTLOOK

In this paper we have shown that the state of the art of language awareness in word processors is not as sophisticated as the language-based support programmers get from their source code editors. Starting from the challenges writers have to face when revising and editing, we developed the concept of language-aware functions for word processors. These functions should be added to existing editors to satisfy the needs of experienced writers, rather than implementing new editors from scratch. We use XEmacs as test bed for our project LingURed. On a general level, the types of language-aware functions for writers and for programmers are very similar, i.e., functions for cursor movement, information functions (including syntax highlighting), and operations. Adapting principles from the implementation of such functions in source code editors is a good starting point for the implementation of analogous functions in word processors.

We have started with the implementation of a selection of language-aware editing functions. Our implementation shows that it is possible to create a variety of helpful functions with only basic NLP resources and through the clever combination of functions already available in XEmacs. We can thus prove two claims: (1) It is possible to implement functions operating on the same level as writers think and talk about texts that help to avoid typical mistakes, and (2) these functions can be implemented with a modest amount of language analysis that mostly operates very locally, i.e., it is often not necessary to parse the entire text written so far. However, the quality of those functions is dependent on the quality of the NLP resources available.

When we started the LingURed project, we were convinced that the time has come to use systems and components from computational linguistics in real-world applications—the computing power today is sufficient to execute even sophisticated processes in reasonable time. Nevertheless, there is room for improvement in terms of quality of results of NLP systems and components. Additionally, computational linguists should work towards freely available resources with well-defined interfaces to allow developers to easily plug a resource into an application function.

We are currently working on implementing further language-aware editing functions as described in section V-E. We are also planning to evaluate the usability and usefulness of the functions with writers from our target group.

Reduction of cognitive load could be evaluated by conducting experiments similar to those described by [31] or [32], i.e, by comparing the products resulting from editing and

revising, and annotating and comparing revisions made in a given text with and without using language-aware functions. If the cognitive load is indeed reduced, the quality of the resulting text should be better and writers should be able to carry out more revisions in a fixed amount of time. In addition, we will interview writers after some of the writing sessions and ask them to fill out questionnaires. We have already conducted a small-scale evaluation of the function for transposing conjuncts; the evaluation results influenced the current implementation and generally showed that writers will use such functions in their daily writing. The acceptance of the function was highly influenced by the fact that it can be called and used in the same way as other XEmacs functions, i.e., that it seamlessly extends the capabilities of the writing environment.

REFERENCES

[1] M. L. Van De Vanter, S. L. Graham, and R. A. Ballance, "Coherent user interfaces for language-based editing systems," *International Journal of Man-Machine Studies*, vol. 37, no. 4, pp. 431–466, October 1992.

[2] M. L. Van De Vanter and M. Boshernitsan, "Displaying and editing source code in software engineering environments," in *Second International Symposium on Constructing Software Engineering Tools (CoSET'2000)*, June 2000.

[3] C. Daiute and R. Taylor, "Computers and the improvement of writing," in *ACM 81: Proceedings of the ACM '81 conference*. New York, NY, USA: ACM, 1981, pp. 83–88.

[4] R. L. Oakman, "The evolution of intelligent writing assistants: trends and future prospects," in *Tools with Artificial Intelligence, 1994. Proceedings., Sixth International Conference on*, 1994, pp. 233–234.

[5] R. Dale, "Computer-based editorial aids," in *Recent Developments and Applications of Natural Language Processing*, J. Peckham, Ed. Kogan Page Limited, 1989, ch. 2, pp. 8–22.

[6] N. Williams, "Writers' problems and computer solutions," *Computer Assisted Language Learning*, vol. 2, no. 1, pp. 5–25, 1990.

[7] P. O. Holt, D. C. Hegg, and T. Johnsen, "Engineering written style," *Computer Assisted Language Learning*, vol. 2, no. 1, pp. 27–35, 1990.

[8] L. R. Taylor, "Software views: A fistful of word-processing programs," *Computers and Composition*, vol. 5, no. 1, pp. 79–90, 1987.

[9] R. M. Collier, "The word processor and revision strategies," *College Composition and Communication*, vol. 34, no. 2, pp. 149–155, 1983.

[10] A. Piolat, "Effects of word processing on text revision," *Language and Education*, vol. 5, no. 4, pp. 255–272, 1991.

[11] D. McCutchen, "A capacity theory of writing: Working memory in composition," *Educational Psychology Review*, vol. 8, no. 3, pp. 299–325, 1996.

[12] A. Piolat, J.-Y. Roussey, T. Olive, and M. Amada, "Processing time and cognitive effort in revision: Effects of error type and of working memory capacity," in *Revision. Cognitive and instructional processes*, ser. Studies in Writing, L. Allal, L. Chanquoy, and P. Largy, Eds. Kluwer Academic Publishers, 2004, vol. 13, pp. 21–38.

[13] D. A. Norman, "Design rules based on analyses of human error," *Commun. ACM*, vol. 26, no. 4, pp. 254–258, 1983.

[14] A. Horning, "Professional writers and revision," in *Revision: History, Theory, and Practice (Reference Guides to Rhetoric and Composition)*, A. Horning and A. Becker, Eds. Parlor Press, 2006, pp. 117–141.

[15] D. Alamargot and L. Chanquoy, "Development of expertise in writing," in *Through the Models of Writing (Studies in Writing, Volume 9)*, ser. Studies in Writing, D. Alamargot and L. Chanquoy, Eds. Kluwer Academic Publishers, 2001, pp. 185–218.

[16] R. M. Stallman, "EMACS the extensible, customizable self-documenting display editor," in *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*. New York, NY, USA: ACM, 1981, pp. 147–156.

[17] M. L. Van De Vanter, *Practical language-based editing for software engineers*, ser. Lecture Notes in Computer Science. Springer, 1995, pp. 251–267.

[18] A. A. Khwaja and J. E. Urban, "Syntax-directed editing environments: issues and features," in *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*. New York, NY, USA: ACM, 1993, pp. 230–237.

[19] D. W. Embley and G. Nagy, "Behavioral aspects of text editors," *ACM Comput. Surv.*, vol. 13, no. 1, pp. 33–70, March 1981.

[20] M. Good, "The use of logging data in the design of a new text editor," *SIGCHI Bull.*, vol. 16, no. 4, pp. 93–97, April 1985.

[21] C. Mahlow, M. Piotrowski, and M. Hess, "Language-aware text editing," in *LREC 2008 Workshop on NLP Resources, Algorithms and Tools for Authoring Aids*, R. Dale, A. Max, and M. Zock, Eds. Marrakech, Morrocco: ELRA, 2008, pp. 9–13.

[22] W. Daelemans and A. van den Bosch, *Memory-Based Language Processing*, ser. Studies in Natural Language Processing. Cambridge, UK: Cambridge University Press, 2005.

[23] K. Koskeniemmi and M. Haapalainen, "GERTWOL – Lingsoft Oy," in *Linguistische Verifikation. Dokumentation zur Ersten Morpholympics 1994*, R. Hausser, Ed. Niemeyer, Tübingen, 1996, ch. 11, pp. 121–140.

[24] C. Mahlow and M. Piotrowski, "Linguistic support for revising and editing," in *Computational Linguistics and Intelligent Text Processing: 9th International Conference, CICLing 2008, Haifa, Israel, February 17–23, 2008. Proceedings*, A. Gelbukh, Ed. Heidelberg: Springer, 2008, pp. 631–642.

[25] R. B. Allen and M. W. Scerbo, "Details of command-language keystrokes," *ACM Trans. Inf. Syst.*, vol. 1, no. 2, pp. 159–178, April 1983.

[26] C. Mahlow and M. Piotrowski, "Opportunities and limits for language awareness in text editors," in *Proceedings of the Workshop on NLP for Reading and Writing – Resources, Algorithms and Tools (SLTC 2008)*, ser. NEALT Proceedings Series, R. Domeij, S. J. Kokkinakis, O. Knutsson, and S. Sofkova Hashemi, Eds., vol. 3. Tartu University Library (Estonia), February 2009, pp. 14–18. [Online]. Available: http://hdl.handle.net/10062/8696

[27] D. Eyman and C. Reilly, "Revising with word processing/technology/-document design," in *Revision: History, Theory, and Practice (Reference Guides to Rhetoric and Composition)*, A. Horning and A. Becker, Eds. Parlor Press, 2006, ch. 7, pp. 102–116.

[28] A. Schiller, S. Teufel, C. Stöckert, and C. Thielen, "Guidelines für das Tagging deutscher Textcorpora mit STTS," Institut für maschinelle Sprachverarbeitung, Stuttgart, Tech. Rep., 1999.

[29] H. Schmid, "Probabilistic part-of-speech tagging using decision trees," in *Proceedings of the International Conference on New Methods in Language Processing*, Manchester, UK, 1994, pp. 44–49.

[30] C. Mahlow and M. Piotrowski, "A target-driven evaluation of morphological components for German," in *Searching Answers – Festschrift in Honour of Michael Hess on his 60th Birthday*, S. Clematide, M. Klenner, and M. Volk, Eds. Münster (Westf), Germany: MV Verlag, 2009, to appear.

[31] L. Faigley and S. Witte, "Analyzing revision," *College Composition and Communication*, vol. 32, no. 4, pp. 400–414, 1981.

[32] R. T. Kellogg, "Components of working memory in text production," in *The Cognitive Demands of Writing: Processing Capacity and Working Memory Effects in Text Production (Studies in Writing, Volume 3)*, ser. Studies in Writing, M. Torrance and G. Jeffery, Eds. Amsterdam University Press, 1999, pp. 42–61.