Detecting and Correcting Morpho-syntactic Errors in Real Texts

Theo Vosse*
Nijmegen Institute for Cognition and Information
University of Nijmegen
and
Cognitive Technology Foundation
P.O. Box 9104
6500 HE Nijmegen, The Netherlands
e-mail: vosse@nici.kun.nl

Abstract

This paper presents a system which detects and corrects morpho-syntactic errors in Dutch texts. It includes a spelling corrector and a shift-reduce parser for Augmented Context-free Grammars. The spelling corrector is based on trigram and triphone analysis. The parser is an extension of the well-known Tomita algorithm (Tomita, 1986). The parser interacts with the spelling corrector and handles certain types of structural errors. Both modules have been integrated with a compound analyzer and a dictionary of 275,000 word forms into a program for stand-alone proof-reading of Dutch texts on a large scale. The system is in its final testing phase and will be commercially available as from 1992.

1. Introduction

One of the most widely used applications of natural language processing is spell, grammar and style checking. Although most probably semantic analysis is required to obtain entirely satisfactory results, it is never used — for obvious reasons. Even worse, most language checkers today even restrain from syntactic analysis. This denies them the possibility to find morpho-syntactic errors, which form a large and frequently occurring class of spelling errors. One of the best known systems for English, which does perform syntactic analysis, is Critique (Richardson, 1988).

In order to detect and correct morpho-syntactic errors a system needs (1) modules for word-level spell checking and correction, (2) a parser which contains a comprehensive grammar and an efficient parsing algorithm, and (3) a mechanism to detect and correct grammatical errors as well as to assist in correcting spelling errors. I will first define the domain of morpho-syntactic errors and motivate the need for a parser. After a brief overview of the system and a discussion of the word-level modules, I will describe the grammar formalism, the parser, its mechanism for error detection, and a pre-processor for word lattices. Finally, after looking at the integration of the modules and at some useful heuristics, I will give a summary of the results obtained by a non-interactive Dutch grammar-driven spell checker.

2. Morpho-syntactic Errors

This paper is concerned with three types of errors: *typographical* errors (typing errors or OCR scanning errors), *orthographical* errors (erroneous transliterations of phonemes to graphemes) and, most importantly, *morpho-syntactic* errors (resulting from misapplication of morphological inflection and syntactic rules). Simple spell checkers are only able to spot errors leading to non-words; errors involving legally spelled words go unnoticed. These morpho-syntactic errors occur quite frequently in Dutch texts, though, and are considered serious because they are seen as resulting from insufficient language competence rather than from incidental mistakes, such as typographical errors. Therefore they constitute an interesting area for grammar checking in office and language teaching applications. I will now present a classification of the morpho-syntactic errors and some related errors in Dutch (Kempen and Vosse, 1990).

2.1. Agreement violations

Typically syntactic errors are agreement violations. Though none of the words in the sentence *She walk home* is incorrect, the sentence is ungrammatical. No simple spelling checking mechanism can find the error, let alone correct it, since it is caused by a relation between two words that need not be direct neighbours. Detection and correction of this type of error requires a robust parser, that can handle ungrammatical input.

*The author's current address is: Experimental Psychology Unit, Leiden University, P.O. Box 9555, 2300 RB Leiden, The Netherlands.

2.2. Homophonous words

Homophony is an important source of orthographical errors: words having the same pronunciation but a different spelling. Dutch examples are *zei* and *zij*, *sectie* and *sexy*, *wort* and *wordt* and *achterruit* and *achteruit*. Such words are easily replaced by one of its homophonous counterparts in written text.

The problem of current spell checkers is that they do not notice this substitution as the substitutes are legal words themselves. In order to detect this substitution, a parser is required since often a change of syntactic category is involved. In section 4.3.2 I will demonstrate that the treatment of these errors strongly resembles the treatment of non-words[1]. Unfortunately, a parser cannot detect substitutions by homophones which have the same syntactic properties.

2.3. Homophonous inflections

A special case of homophonous words are words which differ only in inflection. This type of homophony is very frequent in Dutch and French. French examples are *donner, donnez, donné, donnée, donnés* and *données* or *cherche, cherches* and *cherchent*. Dutch examples typically involve *d/t*-errors: *-d, -t* and *-dt* sound identical at the end of a word but they often signal different verb inflections. Examples are the forms *gebeurt* (third person singular, present tense) and *gebeurd* (past participle) of the verb *gebeuren; word* (first person, singular, present tense) and *wordt* (third person, singular, present tense) of the verb *worden;* and *besteden* (infinitive and plural, present tense),*besteedden* (plural, past tense), and *bestede* (an adjective, derived from the past participle).

However, unlike the general case of homophonous words, homophonous inflections, by their very nature, do not alter the syntactic category of the word but rather its (morpho-syntactic) features. So this type of error can be regarded as a homophonous word or a spelling error, or as an agreement violation.

2.4. Word doubling

Notoriously difficult to spot are word doubling errors, especially at the end of a line ("Did you actually see the the error in this sentence?"). A parser surely notices it, but it should not fail to analyze the sentence because of this.

2.5. Errors in idiomatic expressions

Idiomatic expressions often cause problems for parsers since they often do not have a regular syntactic structure and some of their words may be illegal outside the idiomatic context. A Dutch example is *te allen tijde* (English: *at all times*), with the word *tijde* only occurring in idiomatic expressions. Whenever it occurs in a normal sentence it must be considered to be a spelling error. (An English example might be *in lieu of.*) The problem is even more serious in case of spelling errors. E.g. the expression above is more often than not written as *te alle tijden*, which consists of legal words and is syntactically correct as well.

2.6. Split Compounds

Somewhat similar to idiomatic expressions is the case of compound nouns, verbs, etc. In both Dutch and German these must be written as single words. However, under the ever advancing influence of English on Dutch, many compounds, especially new ones such as *tekst verwerker* (*text processor*) and *computer terminal* are written separated by a blank, thus usually confusing the parser.

3. System overview

The system presented here consists of two main levels: word level and sentence level. Before entering the sentence level (i.e., parsing a sentence), a spelling module should check on all the words in the sentence. This is a rather simple task for a language such as English, but for morphologically complex languages such as Dutch and German, it is by no means trivial. Because compound nouns, verbs and adjectives are written as a single word, they cannot always be looked up in a dictionary, but have to be analyzed instead. There are three problems involved in compound analysis: (1) not every sequence of dictionary words forms a legal compound, (2) certain parts of a compound cannot be found in the dictionary and (3) full analysis usually comes up with too many alternatives. My solution follows the lines set out in (Daelemans, 1987): a deterministic word parser, constrained by the grammar for legal compounds, that comes up with the left-most longest solution first. This solution is rather fast on legal compounds, while it takes at most $O(n^2)$ time for nonexistent words and illegal compounds. The word parser is built upon a simple morphological analyzer, which can analyze prefixes, suffixes and some types of inflection. Both use a dictionary, containing 250,000 word forms[2], derived from 90,000 Dutch lemmata, which appears to be sufficient for most purposes. There is also a possibility to add extra dictionaries for special types of text.

[1]I will not discuss typographical errors resulting in legal words (such as *rotsen* and *rosten*) since their treatment is similar.

[2]For each lemma the dictionary contains all the inflections and derivations that were found in a large corpus of Dutch text (the INL corpus, compiled by the Instituut voor Nederlandse Lexicografie in Leyden). The dictionary itself is a computerised expanded version of the "Hedendaags Nederlands" ("Contemporary Dutch") dictionary, published by Van Dale Lexicografie (Utrecht), which was enriched with syntactic information from the CELEX database (University of Nijmegen).

If a word does not appear in one of the dictionaries and is not a legal compound either, the spell checker can resort to a correction module. In an interactive situation such a module might present the user as many alternatives as it can find. Although this 'the-more-the-better' approach is very popular in commercially available spell checkers, it is not a very pleasant one. It is also unworkable in a batch oriented system, such as the one I am describing here. Ideally, a spelling corrector should come up with one (correct!) solution, but if the corrector finds more than one alternative, it should assign a score or ranking order to each of the alternatives.

The system presented here employs a correction mechanism based on both a variation of trigram analysis (Angell *et al.*, 1983) and triphone analysis (Van Berkel and De Smedt, 1988), extended with a scoring and ranking mechanism. The latter is also used in pruning the search space[3]. Thus the system can handle typographical errors as well as orthographical errors, and includes a satisfactory mechanism for ranking correction alternatives, which is suitable both for interactive environments as well as for stand-alone systems.

When all words of a text have been checked and, if necessary, corrected, a pre-processor (to be described in section 4.4) combines the words and their corrections into a word lattice. The syntactic parser then checks the grammatical relations between the elements in this lattice. If the parsing result indicates that the sentence contains errors, a syntactic corrector inspects the parse tree and proposes corrections. If there is more than one possible correction, it ranks the correction alternatives and executes the top-most one. Section 4 will describe the parser and the pre-processor in some detail. Due to space limitations, I have to refer to (Vosse, 1991) for further information, e.g. the adaptations that need to be made to the Tomita algorithm in order to keep the parsing process efficient.

4. Shift-Reduce Parsing with ACFGs

4.1. Augmented Context-free Grammars

Augmented Context-Free Grammars (ACFGs for short) form an appropriate basis for error detection and correction. Simply put, an ACFG is a Context-Free Grammar where each non-terminal symbol has a (finite) sequence of attributes, each of which can have a set of a finite number of symbols as its value.

[3]Pruning the search space is almost obligatory, since trigram and triphone analysis require $O(n*m)$ space, where n is the length of the word and m the number of entries in the dictionary. The constant factor involved can be very large, e.g. for words containing the substring ver, which occurs in more than seven out of every hundred words (13,779 triphones and 16,881 trigrams in 237,000 words).

In a rule, the value of an attribute can be represented by a constant or by a variable.

A simple fragment of an ACFG is for example:

```
1 S → NP(Num nom) VP(Num)
2 NP(Num _) → Det(Num) ADJs Noun(Num)
3 NP(Num Case) → Pro(Num Case)
4 VP(Num) → Verb(Num intrans)
5 VP(Num) → Verb(Num trans) NP(_ acc)
6 ADJs → ε
7 ADJs → ADJ ADJs
```

The derivation of a sentence might go like this:

```
S ⇒ NP(sg3 nom) VP(sg3) ⇒ Det(sg3) ADJs
Noun(sg3) VP(sg3) ⇒ Det(sg3) Noun(sg3) VP(sg3)
⇒ Det(sg3) Noun(sg3) Verb(sg3 intrans) ⇒ a
man eats
```

In the actual implementation of the parser, the grammatical formalism is slightly more complex as it uses strongly typed attributes and allows restrictions on the values the variables can take, thereby making grammar writing easier and parsing more reliable. The Dutch grammar employed in the system contains nearly 500 rules.

4.2. The parser

The construction of the parsing table is accomplished by means of standard LR-methods, e.g. SLR(0) or LALR(1), using the "core" grammar (i.e. leaving out the attributes). The parsing algorithm itself barely changes as compared to a standard shift-reduce algorithm. The shift step is not changed except for the need to copy the attributes from lexical entries when using a lexicon and a grammar with pre-terminals. The reduction step needs to be extended with an instantiation algorithm to compute the value of the variables and a succeed/fail result. It should fail whenever an instantiation fails or the value of a constant is not met.

To accomplish this, the trees stored on the stack should include the values resulting from the evaluation of the right-hand side of the reduced rule. This makes the instantiation step fairly straightforward. The variables can be bound while the elements are popped from the stack. If a variable is already bound, it must be instantiated with the corresponding value on the stack. If this cannot be done or if a constant value in a rule does not match the value on the stack, the reduction step fails. A simple example (not completely) following the grammar sample above may clarify this.

In Figure 1a parsing succeeds just as it would have done if only the context-free part of the grammar had been used. The only difference is that the symbols on the stack have attributes attached to them. In Figure 1b however, parsing fails — not because the context-free part of the grammar does not accept the sentence (the parse table does contain an entry for this case) but because the instantiation of pl and sg3 in rule 1 causes the reduction to fail.

Note that the mechanism for variable binding is not completely equivalent to unification. It typically differs from unification in the reduction of the following two rules

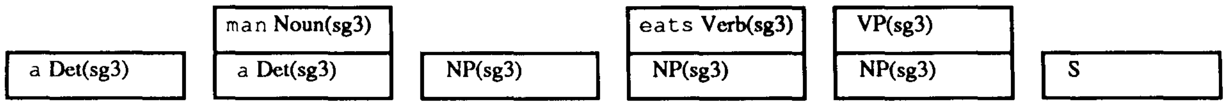

Figure 1a. Parsing of "a man eats".

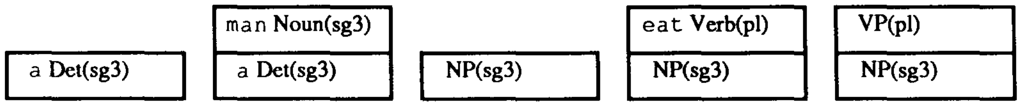

Figure 1b. Parsing of "a man eat"

```
1 A → ... B(X, Y) ...
2 B(X, X) → ...
```

The reduction of rule 2 will leave two values on the stack rather than an indication that the two variables are one and the same. Therefore X and Y may differ after the reduction of rule 1.

4.3. Parsing Erroneous Input

4.3.1. Coercing syntactic agreement

Figure 1b shows one type of problem I am interested in, but clearly not the way to solve it. Though the parser actually detects the error, it does not give enough information on how to correct it. It does not even stop at the right place[4], since the incongruity is only detected once the entire sentence has been read. Therefore the reduction step should undergo further modification. It should not fail whenever the instantiation of a variable fails or a constant in the left-hand side of the rule being reduced does not match the corresponding value on the stack, but mark the incongruity and continue parsing instead. Later in the process, when the parsing has finished, the syntactic corrector checks the marks for incongruity and coerces agreement by feature propagation.

This approach contrasts with, e.g., the approach taken by (Schwind, 1988), who proposes to devise an error rule (cf. section 4.3.3) for every unification error of interest. However, this makes efficient parsing with a large grammar nearly impossible since the size of the parsing table is exponentially related to the number of rules.

4.3.2. Syntactic filtering

Consider the error in *The yelow cab stops*. The English spelling corrector on my word processor (MS-Word) offers two alternatives: *yellow* and *yellows*. Since the string *yelow* is obviously incorrect, it has no syntactic category and the sentence cannot be parsed. One might therefore try to substitute both alternatives and see what the parser comes up with, as in Figure 2. This example clearly shows that the only grammatically correct alternative is *yellow*. In this way a parser can help the spelling corrector to reduce the set of correction alternatives. Since a realistic natural language parser is capable of parsing words with multiple syntactic categories (e.g. *stop* is both a noun and a verb), the two entries for yelow can be parsed in a similar fashion. The grammatical alternative(s) can be found by inspecting the resulting parse trees afterwards.

In order to handle errors caused by homophones as well, this mechanism needs to be extended. When dealing with legal words it should use their syntactic categories plus the syntactic categories of all possible homophones, plus — to be on the safe side — every alternative suggested by the spelling corrector. Afterwards the parse trees need to be examined to see whether the original word or one of its alternatives is preferred.

4.3.3. Error rules

The third and last category of errors the system attempts to deal with consists of the structural errors. General techniques for parsing sentences containing errors are difficult, computationally rather expensive and not completely fool-proof. For these reasons, and because only a very limited number of structural errors occur in real texts, I have developed a different approach. Instead of having a special mechanism in the parser find out the proper alternative, I added error rules to the formalism. The grammar should now contain foreseen improper constructions. These might treat some rare constituent order problems and punctuation problems.

4.3.4. Parsing weights

Natural language sentences are highly syntactically ambiguous, and allowing errors makes things considerably worse. Even the simple toy grammar above yields a great number of useless parses on the sentence *They think*. The word think may have different entries for 1st and 2nd person singular, 1st, 2nd and 3rd person plural and for the infinitive. This

[4]This of course is caused by the context-free part of the grammar. If we had created a unique non-terminal for every non-terminal-feature combination, e.g. `S -> NP_sing3_nom VP_sing3`, parsing would have stopped at the right place (i.e. between "man" and "eat"). This however depends mainly on the structure of the grammar. E.g. in Dutch the direct object may precede the finite verb, in which case agreement can only be checked after having parsed the subject following the finite verb. Then the parser cannot fail before the first NP following the finite verb. This is too late in general.

```
the yellow cab stops
 0
 1 Det     0
13 yellow  1 Det  0
11 ADJ     1 Det  0
20 ADJs   11 ADJ  1 Det 0
12 ADJs    1 Det  0
21 Noun   12 ADJs 1 Det 0
 2 NP      0
15 Verb    2 NP   0
14 VP      2 NP   0
 4 S       0
   Accept
```

```
the yellows cab stops
 0
 1 Det      0
12 ADJs      1 Det  0
25 Noun     12 ADJs 1 Det 0







    Fails
```

Figure 2. The parsing of the two alternatives for "the yelow cab stops".

would result in one parse tree without an error message and five parse trees indicating that the number of *they* does not agree with the number of *think*. By using sets of values instead of single values this number can be reduced, but in general the number of parses will be very large. Especially with larger grammars and longer sentences there will be large amounts of parses with all sorts of error messages.

A simple method to differentiate between these parses is to simply count the number of errors, agreement violations, structural errors and spelling errors in each parse, and to order the parses accordingly. Then one only has to look at the parse(s) with the smallest number of errors. However, this concept of weight needs to be extended since not all errors are equally probable. Some types of agreement violation simply never occur whereas others are often found in written texts. Orthographical and typographical errors and homophone substitution are frequent phenomena while structural errors are relatively rare. Suppose the parser encounters a sentence like *Word je broer geopereerd?* (Eng.: *Are your brother (being) operated?*). In Dutch this is a frequent error (see section 2.3), since the finite verb should indeed be *word* if *je* instead of *je broer* were the subject. (Translating word-by-word into English, the correction is either *Is your brother (being) operated?* or *Are you brother (being) operated? Je* is either *you* or *your*.) The most likely correction is the first one. How can a syntactic parser distinguish between these two alternatives? My solution involves adding error weights to grammar rules. These cause a parse in which verb transitivity is violated to receive a heavier penalty than one with incorrect subject verb agreement. Thus, parse trees can be ordered according to the sum of the error weight of each of their nodes.

4.4. Word Lattices

As noted in section 2.5, idiomatic expressions cause parsers a lot of trouble. I therefore propose that the parser should not operate directly on a linear sentence, but on a word lattice that has been prepared by a pre-processor. For a sentence like *Hij kan te allen tijde komen logeren* (*he can come to stay at all times*) such a structure might look like Figure 3. Instead of parsing each word of the expression *te allen tijde* separately, the parser can take it as a single word spanning three word positions at once or as three separate words. Should one of the words in the expression have been misspelled, the pre-processor builds a similar structure, but labels it with an error message containing the correct spelling obtained from the spelling corrector. Word lattices can of course become much more complex than this example.

Since there is a pre-processor that is able to combine multiple words into a single item, it might as well be used to aid the parser in detecting two further types of errors as well. The first one is the Dutch split compound. By simply joining all the adjacent nouns (under some restrictions) the grammar and the parser can proceed as if split compounds do not occur. The second error type is word doubling. The pre-processor can join every subsequent repetition of a word with the previous occurrence so that they will be seen both as two distinct words and as one single word (since not every occurrence of word repetition is wrong). Another possibility is to concatenate adjacent words when the concatenated form occurs as one entry in the dictionary. E.g. many people do not know whether to write *er op toe zien, erop toezien, er op toezien* or any other combination (though a parser might not always have the right answer either).

5. Integration and Heuristics

The combination of the modules described above — a spell checker with compound analysis, a spelling corrector, a robust parser and a syntactic corrector — does not lead by itself to a batch-oriented proof-reading system. Most texts do not only contain sentences, but also titles and chapter headings, captions, jargon, proper names, neologisms, interjections, dialogues (*"yes", she said, "yes, that is true, but..."*), quotations in other languages, literature references, et cetera, not to mention mark-up and typesetting codes. The system therefore has a mechanism for dealing with the layout aspects of

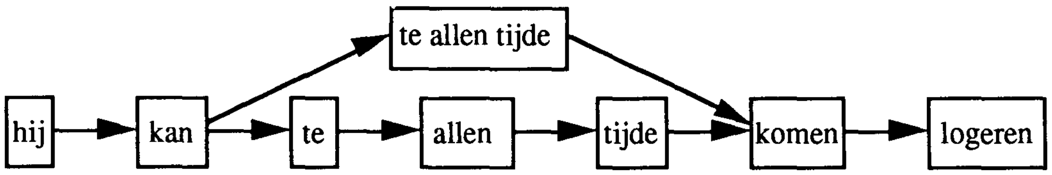

Figure 3. A word lattice.

texts and some heuristics for dealing with proper names, jargon and neologisms. The layout aspects include mark-up codes and graphics, title markers and a mechanism for representing diacritics, such as the diaeresis, which is frequent in Dutch.

Dictionaries seldom contain all words found in a text. In Dutch, part of the problem can be solved by using compound analysis. However, a misspelled word can sometimes be interpreted as a compound, or as two words accidentally written together. I partially solved this problem by having the compound analyzer repeat the analysis *without* the word grammar if it fails *with* the word grammar, and by defining a criterion which marks certain compounds as "suspicious"[5]. If the analyzer marks the compound as either suspicious or ungrammatical, the spelling corrector is invoked to see if a good alternative (i.e. closely resembling and frequent word) can be found instead, or, else, if the compound was ungrammatical, whether it can be split into separate words. This process is further improved by adding the correct compounds in the text to the internal word list of the spelling corrector.

Other words that do not appear in a dictionary are proper names, jargon and neologisms. Therefore the system first scans the entire text for all word types while counting the tokens before it starts parsing. My rule of thumb is to treat words, that appear mainly capitalized in the text as proper names. Frequently occurring words, that do not have a good correction, are supposed to be neologisms. Both proper nouns and neologisms are added to the internal word list of the spelling corrector. The main disadvantage of this approach is that it misses consistently misspelled words. At the end of the run therefore, the system provides a list of all the words it tacitly assumed to be correct, which must then be checked manually.

Another feature of the system is that it coerces variant spelling into preferred spelling. This feature also takes compounds which have is no official preferred spelling into consideration, thus preventing compound to be written in different ways. E.g. both *spellingcorrectie* and *spellingscorrectie* (Eng.: *spelling correction*) are correct in Dutch. My system only allows one to occur in a text and coerces the least frequently occurring variants into the most frequent one.

The last but not least important tricks help to reduce parsing time. Since the system cannot detect all types of errors with equal reliability (cf. section 6), I added a *d/t*-mode in which only sentences that *might* contain a *d/t*-error (cf. section 2.2) are parsed. In this mode a pre-processor first checks whether the sentence contains such a "*d/t*-risk" word. If this is the case the parser is invoked, but the error messages not pertaining to this class of errors are suppressed. As *d/t*-risks show up in less than a quarter of all sentences, parsing time is cut by a factor of four at least. Although this solution can hardly be called elegant, it gives the user a faster and more reliable system.

There also is an upper bound on the number of allowed parses. Because analyzing a parse tree takes some time, this speeds up the process. The disadvantage is that the system may choose an unlikely correction more often as it cannot compare all parse trees. Large sentences with multiple errors may produce thousands of parse trees, each of which has to be scored for comparison. As the allowed number of parses becomes less than the potential number of parses, the probability that the system overlooks a likely correction grows. But since it produces an error message anyway, albeit an unlikely one, the advantage outweighs the disadvantage.

6. Results and Evaluation

The system described in this paper has been built as a practical writing aid that operates non-interactively, because the first phase (determining word types, compound analysis, initial spelling correction, and cross-checking corrections for the entire text) takes too long. Nevertheless, it can easily process more than 25 words per second[6] for a large text, which may easily take up half an hour or more.

As an example of the performance in the *word level* checking phase, I presented the system with a

[5]Misspelled word can often be analyzed as sequences of very small words. E.g. the misspelled *kwaliteitesverbetering* (which should be *kwaliteitsverbetering*, Eng.: *quality improvement*) can be divided into *kwaliteit+es+verbetering*, which could mean *quality ash improvement*. The amount of overgeneration correlates strongly with the size of the dictionary.

[6]I have written the system in the programming language C. The results reported below were obtained with the program running on a DECstation 3100. Part of the speed derives from the frequent repetition of many words in large texts.

random sample of 1000 lines from two large texts[7]. The sample contained nearly 6000 words, with 30 true spelling errors. Of these, 14 were corrected appropriately, and 14 were found but substituted by an incorrect alternative or not corrected at all. Of the 14 appropriately corrected errors, 9 were errors in diacritics only. The system only missed 2 errors, which it assumed to be proper names (both reported at the end of the file (cf. section 5)). It also produced 18 false alarms, 11 of which were caused by very infrequent jargon or inflected word forms missing from the dictionary.

Comparison with other spell checkers is hardly possible. For Dutch, only elementary spell checkers based on simple word lookup are available. If this method is applied to the sample text with the same dictionary as used in the full system, the result is entirely different. Such a simple spell checker marks 217 words as misspelled. Among these are not only the 21 true errors and the 9 errors wrongly placed diacritics, but also 37 abbreviations and proper names, and 150 compounds. This amounts to a total of 187 false alarms!

The *sentence level* requires considerably more time. Error-free short sentences can be parsed at a speed of four or more words per second, but long sentences containing one or more errors may require several seconds per word (including correction, which is also rather time consuming). For the texts mentioned in footnote 7 (110,000 words in total), the CPU time required for parsing was approximately 7 hours.

But what counts is not only speed; quality is at least equally important. Preliminary tests have shown satisfactory results. A 150 sentence spelling test for secretaries and typists, with an average sentence length between six and seven, was performed within nine minutes (elapsed time) leaving only three errors undetected, correcting the other 72 errors appropriately and producing no false alarms. (Human subjects passed the test if they could complete it within ten minutes making at most ten mistakes.) The three undetected errors involved semantic factors, and were therefore beyond the scope of the system. The rightly corrected errors were typographical and (mainly) orthographical errors, agreement errors and errors in idiomatic expressions.

Other spelling exercises also showed good results (most errors detected and most corrected properly, very few false alarms, if any). A typical text was chosen from a text book with correction exercises for pupils. In contrast with the spelling test described in the previous paragraph, most sentences in this test contained more than one spelling error. The errors varied from superfluous or missing diaeresis to split compounds and *d/t*-errors. On a total of 30 sentences, the system discovered 75 errors, of which 62 were corrected properly, 12 miscorrected and one was given no correction at all; it missed 7 errors, while producing one false alarm. Although the total number of words was only half the number of words in the previous test (457 to be precise), the system took almost three times as much time to process it. This was partly due to the greater average sentence length (over 15 words per sentence) and the occurrence of more than one error per sentence (up to four per sentence). The number of errors that could not have been detected without a parser was 18. Of these, 10 were corrected and 1 was detected but substituted by a wrong alternative, while the parser missed the 7 errors mentioned earlier.

On large real texts, i.e. not constructed for the purpose of testing one's knowledge of spelling, the system performed less well due to parsing problems. As an example of a well written text, I took the first 1000 lines of a text mentioned in footnote 7. This sample consisted of 7443 words in 468 sentences (an average of nearly 16 words per sentence). At word level it performed quite satisfactorily. It caused 12 false alarms[8], while detecting 11 true errors, of which only 4 were properly corrected. The compound analysis functioned almost flawlessly. However, it caused 6 of the 12 false alarms, because one single word, which was not in the dictionary, appeared in 4 different compounds. The heuristics for suspicious words cooperated very well with the spelling corrector (6 correct guesses, 2 wrong).

The parser's performance however degraded considerably. One reason was the great length of many sentences (up to 86 words). This sometimes caused the parser to exceed its built-in time limit, so that it could not give a correct error message[9]. Long sentences are also highly ambiguous. This increases the probability of finding a very awkward but error-free parse, thereby overlooking real errors. Another reason for the performance degradation was the abundant use of interjections, names (between quotes, dashes or parentheses) and colloquial (ungrammatical) expressions. Although the parser has some provisions for simply skipping such con-

[7]These manuscripts are representative for texts submitted to the system by a publisher who has access to it. A typical example is a text concerning employment legislation and collective wage legislation of over 660,000 characters (a total of 92,000 words) of plain text with mark-up instructions. Checking the words and correcting misspelled words took 16 CPU minutes, which results in a speed of nearly 100 words per CPU second. A smaller text in the same content domain (150,000 characters in 27,500 words) was checked and corrected at word level in 4.5 minutes of CPU time, which is over 100 words per CPU second.

[8]In 4 cases, the false alarm was caused by word contraction. E.g. the word *echtgeno(o)t(e)*, which is supposed to mean *echtgenoot of echtgenote* (*husband or wife*), was marked incorrect and substituted by *echtgenoot*.

[9]Unfortunately, the program does not keep track of this, so no data can be specified.

structions, they more often than not interfere with error detection. Fortunately, subject-verb agreement errors indicating *d/t*-errors were spotted quite reliably, although their number (two in this sample, which were both corrected) is too small to draw any firm conclusion. The detection of punctuation errors and split compounds still needs improvement. Whether the results justify the 30 minutes CPU time it took to parse the 468 sentences remains to be seen.

7. Conclusions

I have shown the feasibility of building a practical grammar-based spell checker that detects and corrects the important class of morpho-syntactic errors in normal texts (i.e., texts that have not been specially prepared before processing). The system described in this paper is the first example of such a spell checker for Dutch. It is currently being tested at a large publishing company.

I have demonstrated what can be expected of the approach I have taken. Depending on the complexity of the sentences, the combination of a word-level spell checker plus a syntactic parser performs from nearly perfect to satisfactory in regard to morpho-syntactic errors. Other types of errors cannot be handled reliably with the current framework, partly due to the permissive nature of both grammar and dictionary. However, enrichment of grammar and lexicon is only possible on an ad hoc basis. It will not lead to a systematic improvement of the correction process. Moreover, it is likely to interfere with the other components. Although many details still have to be worked out, the limits of this approach become visible. The next major improvement must come from analysis beyond syntax.

Acknowledgements

The author would like to thank (in random order): Edwin Bos for commenting on earlier versions of this paper; Alice Dijkstra, for her work on this project and for proof-reading and commenting on this paper; and Gerard Kempen, project leader, for discussions and comments.

References

Angell, R.C., G.E. Freund, P. Willet. 1983. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management* (19), pp. 255-261.

Berkel, Brigitte van, and Koenraad de Smedt. 1988. Triphone analysis: a combined method for the correction of orthographical and typographical errors. In: *Proc. 2nd Conference on applied natural language processing*. Association for Computational Linguistics, pp. 77-83.

Daelemans, W. 1987. *Studies in language technology: an object-oriented model of morpho-phonological aspects of Dutch*. Ph. D. dissertation, University of Leuven.

Kempen, Gerard, and Theo Vosse. 1990. A language sensitive editor for Dutch. In: *Proc. Computer & Writing III Conference*, Edinburgh.

Nakazawa, Tsuneko. 1991. An extended LR parsing algorithm for grammars using feature-based syntactic categories. In: *Proc. 5th Conference of the European chapter of the ACL*, Berlin. pp. 69-74.

Richardson, S.D. 1988. The experience of developing a large-scale natural language text processing system: CRITIQUE. In: *Proc. 2nd Conference on Applied Natural Language Processing*. Association for Computational Linguistics.

Schwind, Camilla. 1988. Sensitive parsing: error analysis and explanation in an intelligent language tutoring system. In: *Proc. COLING '88*, Budapest, pp. 608-613.

Tomita, Masaru. 1986. *Efficient parsing for natural language: a fast algorithm for practical systems*. Dordrecht, Kluwer.

Vosse, Theo. 1991. Detection and correction of morpho-syntactic errors in shift-reduce parsing. In: *Tomita's Algorithm: Extensions and Applications* . R. Heemels, A. Nijholt, K. Sikkel (Eds.). Memoranda Informatica 91-68, Univ. of Twente, 1991, pp. 69-78.