Grammar Checking with Dependency Parsing: A Possible Extension for LanguageTool

Maxim Mozgovoy
University of Aizu, Tsuruga, Ikki-machi, Aizu-Wakamatsu, Fukushima, 965-8580 Japan
E-mail: mozgovoy@u-aizu.ac.jp

Keywords: grammar checking, dependency parsing, LanguageTool

Received: October 25, 2011

This paper describes a possible extension for a well-known open source grammar checking software LanguageTool. The proposed extension allows the developers to write grammatical rules that rely on natural language parser-supplied dependency trees. Such rules are indispensable for the analysis of word-word links in order to handle a variety of grammatical errors, including improper use of articles, incorrect verb government, and wrong word form agreement.

Povzetek: Članek opisuje razširitev programa LanguageTool s pravili, ki uporabljajo odvisnostna drevesa.

1 Introduction

Grammar checking is a well-recognized problem of natural language processing. Grammar checkers are helpful in a variety of scenarios, such as text authoring and language learning. The purpose of such tools is to find grammatical errors in the input text: incorrect use of person, number, case or gender, improper verb government, wrong word order, and so on. A grammar checker normally works in combination with a spellchecker — a module that detects spelling errors in individual words. As a rule, spell checker cannot correct even basic grammatical flaws, such as erroneous choice of article (like in the expression “an box”).

While a spellchecker is already an essential part of a modern text authoring system, a grammar checking module is still found only in large commercial packages like Microsoft Office or WordPerfect Office. Certain grammar checkers are also available as additional software packages or online services, offered by independent companies [1-3].

This situation is slowly changing nowadays. With the growing popularity of open source software, more natural language processing systems should become available for wider use. Open spellchecking libraries, such as JOrtho and GNU Aspell already exist, and anyone can extend own software with their capabilities. Grammar checking is a more challenging task, and most open projects are still far beyond well-established proofing tools, such as offered by MS Word.

*Supported in part by the Fukushima Prefectural Foundation, Project F-23-1, FY2011.

This paper is based on M. Mozgovoy, *Dependency-Based Rules for Grammar Checking with LanguageTool* published in the proceedings of the 1st International Workshop on Advances in Semantic Information Retrieval (part of the FedCSIS’2011 conference).

1.1 Rule-Based Grammar Checking

Probably, the predominating approach to grammar checking today consists in testing the input text against a set of handcrafted rules [4, 5]. For example, the rule

```
I + Verb (3rd person, singular form)
```

corresponds to the incorrect verb form use, as in the phrase “I has a dog”. In order to emphasize the nature of such rules as erroneous patterns, they are often called “mal-rules”.

This method has several attractive features: (a) rules can be easily added, modified or removed; (b) every rule can have a corresponding extensive explanation, helpful for the end user; (c) the system is easily debuggable, since its decisions can be traced to a particular rule; (d) the rules can be authored by the linguists, possessing limited or no programming skills. An obvious disadvantage of a rule-based system is a large amount of manual work, needed to build an extensive rule set.

An alternative approach is represented with several varieties of statistical systems that analyze existing collections of grammatically correct and incorrect texts, attempting to find word patterns and/or text features that correspond to correct sentences [6, 7]. The simplest statistical grammar algorithm consists in analyzing N-grams — chains of N consecutive words [8]. If a certain word chain is common in the master text corpus, it is considered correct.

Statistical grammar checkers have their own advantages and drawbacks, but their analysis is beyond the scope of this article.

1.2 Introducing LanguageTool

The purpose of the present work is to design a possible extension for the LanguageTool grammar checker [9]. LanguageTool is a modern rule-based open source

grammar checking system, available both as a plug-in for OpenOffice.org and as a downloadable library, which makes it ready for use in any software projects. Currently LanguageTool supports 21 languages, though the number of ready grammatical rules ranges from 4 for Lithuanian to 1994 for French (as of November, 2011). The rules can be authored by any interested contributors.

Unfortunately, the syntax of rules in LanguageTool does not allow formulating certain grammatical phenomena. In the subsequent sections, we will consider these limitations and a possible method to reduce them.

2 Basic Design Principles of LanguageTool

LanguageTool defines an XML-based language for describing mal-rules. In its simplest form, a mal-rule is just a sequence of tokens to be matched in the text:

```
<!-- "all be it" instead of "albeit" -->

<pattern>
  <token>all</token>
  <token>be</token>
  <token>it</token>
</pattern>
<message>Did you mean 'albeit'?</message>
```

The syntax of the rules is flexible and powerful: it is possible to use OR and NOT logic operations (“match token A or token B”; “match any token except C”), to skip optional tokens, and, to some extent, to use regular expressions.

Several syntactic elements are backed with additional linguistic modules — *sentence splitter* and *part-of*-speech *tagger*. Sentence splitter determines the boundaries of each sentence, thus allowing the user to find certain tokens exactly at the beginning or at the end of a sentence:

```
<!-- "another words," instead of
     "in other words,"
     at the beginning of a sentence -->

<pattern>
  <token postag="SENT_START"></token>
  <token>another</token>
  <token>words</token>
  <token>,</token>
</pattern>
<message>Did you mean
         'in other words'?</message>
```

Part-of-speech tagger determines every word’s part of speech, helping the user to find tokens that belong to a certain class:

```
<!-- "ca" + [personal pronoun] instead of
     "can" + [personal pronoun] -->

<pattern>
  <token>ca</token>
  <token postag="PRP"></token>
</pattern>
<message>Did you mean 'can'?</message>
```

LanguageTool makes use of third-party libraries for splitting and tagging the input text. Fortunately, a number of ready solutions are available for this purpose (e.g., Ratnaparkhi’s MXTERMINATOR and MXPOST [10, 11]).

3 Introducing Dependency-Based Rules

Despite high expressive power and flexibility, LanguageTool’s rule system has a notable shortcoming: it treats the input text as a sequence of tokens, ignoring tree-like nature of natural language sentences.

Consider, for example, the following problem. In English, `a/an` article should never be used with a noun in a plural form. The current LanguageTool rule to detect such a case is defined as follows:

```
<!—"a/an" article, then a plural noun -->

<pattern>
  <token regexp="yes">a|an</token>
  <token postag="NNS|NNPS"</token>
</pattern>
<message>Don't use indefinite articles
         with plural words.</message>
```

However, this rule ignores the fact that there can be any number of words between `a/an` and the corresponding noun (“a box”, “a wooden box”, “a simple wooden box”). The rule definition can be improved if we allow any number of optional adjectives between the article and the noun, but in general case this solution is inadequate.

In order to handle such problems, the grammar checker should analyze nonlinear structure of the phrase. An article is logically linked with a noun, regardless of any words between them. This nonlinear structure can be obtained with an additional module, known as *dependency parser*. This instrument represents the structure of every sentence with a *parse tree*, having words as nodes and logical links between them as edges (see Fig. 1).

As it can be seen, the article “a” is linked directly to the noun “box”. Having such a tree, it is possible to extend the syntax of LanguageTool grammatical rules, enabling the developers to analyze word-word relationships.

4 Technical Approach

In order to achieve our goals, we had to solve three subproblems: 1) select a suitable dependency parsing instrument; 2) develop an appropriate syntax for dependency-based rules; 3) design the corresponding rule-matching algorithm.

4.1 Selecting a Practical Dependency Parser

After examining currently available solutions, we decided to use one of two parsers: MaltParser [12] or

LDPar [13]. Both of them are high-quality dependency parsers, available as open source.

MaltParser is written in Java, and thus suits better for the use in combination with the current implementation of LanguageTool, also made with Java. LDPar distribution contains cross-platform C++ code, providing compilable efficient implementation. Both parsers are based on machine learning: the parser first has to be trained with a collection of correctly parsed sentences (a *treebank*). MaltParser and LDPar also share the same format of input and output data.

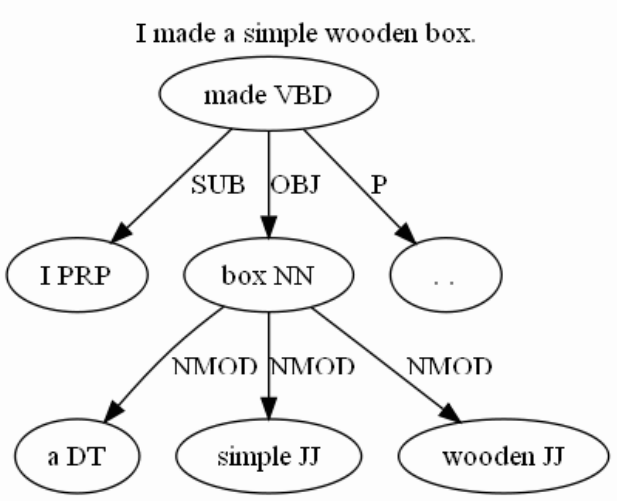

Figure 1: Parse tree for the phrase "I made a simple wooden box."

4.2 Syntax for Dependency-Based Rules

Dependency-based rules should provide syntactic means for the following basic functions:

1) Match a link between two given words, optionally labeled with a given label. This function should be generalizable to the matching of the whole subtree.
2) Check whether a certain word appears before or after another word, in order to control word precedence.
3) Check for the absence of the given subtree in the parse tree.

In order to satisfy these requirements, we suggest the following syntax for an individual dependency-based rule. The rule definition is split into chunks, each representing a separate subtree to be matched:

```
CHUNK1
CHUNK2
...
CHUNKN
```

Every `CHUNKi` is represented with a sequence of tokens, defined with `token` XML tag:

```
<token [attributes]>token-value</token>
```

Currently our system supports the following attributes:

- **pos="text"**: the token should belong to the specified part-of-speech class;
- **label="text"**: the link to the token's parent (according to the parse tree) should has the specified label;
- **parent="number"**: the token should have the specified token as a parent (according to the parse tree);
- **except**: the token's value should not match `token-value`;
- **before="number"**: the token should appear in the sentence before the specified token;
- **after="number"**: the token should appear in the sentence after the specified token;
- **chunk_start**: start-of-chunk marker;
- **inverse**: the current chunk (subtree) should not be found in the parse tree;
- **set_anchor="text"**: the token will be marked with a symbolic "anchor" (see below);
- **anchor="text"**: the token should be found at the specified anchor position in the parse tree.

Attributes **parent**, **before**, and **after** expect a token's cardinal number within the current chunk as an argument. By default, every chunk of the rule has to be matched in the parse tree in order to satisfy the rule.

Anchors are introduced to simplify the analysis of the subtrees. For example, suppose that the first chunk of a rule matches an object of the sentence's root verb:

```
<token label="ROOT"></token>
<token parent="1" label="OBJ"
       set_anchor="anchor1"></token>
```

Now, suppose that a certain chunk later in the chain needs to check that this object has dependent adjectives. By using the anchor, it is possible to start matching directly from the right place:

```
<token anchor="anchor1"></token>
<token parent="1" pos="ADJ"></token>
```

4.3 Examples

The following examples illustrate the capabilities of dependency-based rules:

```
<!-- Example 1:
     in non-interrogative sentences
     the subject should be placed before
     the predicate -->

<token pos="VB|VBP|VBZ|VBD"
       label="ROOT"></token>
<token after="1" label="SUB"></token>

<token chunk_start="" inverse=""
       label="ROOT"></token>
<token parent="1">?</token>
```

The first chunk ensures that the system has found a subject (labeled SUB), placed after the main verb. The second chunk asserts the absence of '?' mark, linked to the tree root.

```
<!-- Example 2:
     "a/an" should not be used
     with plural nouns -->

<token>a|an</token>
<token pos="NNS|NNPS" parent="1"></token>
```

This mal-rule finds a/an articles, linked to plural nouns (marked as NNS or NNPS by a part-of-speech tagger). Note that the determiner (such as an article) is always directly connected to the corresponding word, even if they are not adjacent in the original sentence.

```
<!-- Example 3:
     the gerund should be used in
     conjunction with auxiliary verbs -->

<token pos="VBG" label="ROOT"></token>
```

If a gerund (verb ing-form) is recognized as a parse tree root, this means the absence of an obligatory auxiliary verb (such as "is", "was"). If an auxiliary verb is present, it becomes a root element of the tree.

```
<!-- Example 4:
     improper personal verb form used -->

<token pos="VBZ"></token>
<token parent="1"
       label="SUB">I|we|you|they</token>
```

If the subject of a certain verb is I/we/you/they, the verb should not be in the 3rd person singular form.

4.4 Implementation

Each chunk of a rule is matched separately. The chunk-matching algorithm is a straightforward implementation of the depth-first search routine:

```
// token_index : integer
// used_tokens: set of integers
bool matchSubtree(token_index,
                  used_tokens)
{
  if(token_index > MAX_INDEX_IN_CHUNK)
    return true;

  for_each(token k in the tree)
    if(used_tokens.contains(k) == false)
      if(tokens_match(token_index, k)
        if(matchSubtree(token_index + 1,
                  union(used_tokens, k))
          return true;

  return false;
}

// first call:
// b = matchSubtree(0, empty());
```

5 RuleDesigner Tool

In order to simplify rule authoring, we have also created a specialized RuleDesigner tool that provides a centralized interface for the development, debugging and testing of grammatical rules.

Our approach to the rule creation process can be compared with test-driven development. Each rule has a special "self-tests" section that contains an arbitrary text fragment. Self-test is passed if the system finds the given number of errors in this text, using the current rule:

```
<test matches="3">I loves London.
We eats in London.
John and I loves London.</test>
```

RuleDesigner automatically runs self-tests and displays all the information needed to check and correct grammatical rules:

- editable rule definition;
- editable list of self-tests;
- the results of self-tests.

Furthermore, for each sentence in self-tests RuleDesigner shows its parse tree[1], part-of-speech markup, explanation to the user, and a list of possible text corrections[2] (see Figure 2).

It is also possible to run tests on a user-specified text block with all the rules turned on. It helps to identify false positives, not revealed with isolated rule-level self-tests.

6 Discussion

LanguageTool is a good example of an extensible rule-based grammar checker. Basic grammatical rules can be expressed by means of standard regular expressions. If their expressive power is insufficient to describe a certain rule, one can make use of additional natural language processing-powered syntactic elements, backed with sentence splitter and part-of-speech tagger.

This architecture can be extended further by incorporating other language processing modules. An obvious candidate for this role is a natural language parser that shows immediate word-word relationships. We have demonstrated several examples of grammatical errors, detectable with parser-powered mal-rules.

Since we consider rule-based grammar checking to be an established technology, the discussion of its advantages and drawbacks is beyond the scope of our work. However, our experiments have revealed weak points of the language tools we use (parser and part-of-speech tagger, mainly).

Normally, these tools, being based on machine learning algorithms, need initial training on annotated text data. Most such training collections are represented

[1] Obtained with AT&T GraphViz tool.

[2] Discussion of text-correction functionality is outside the scope of this paper.

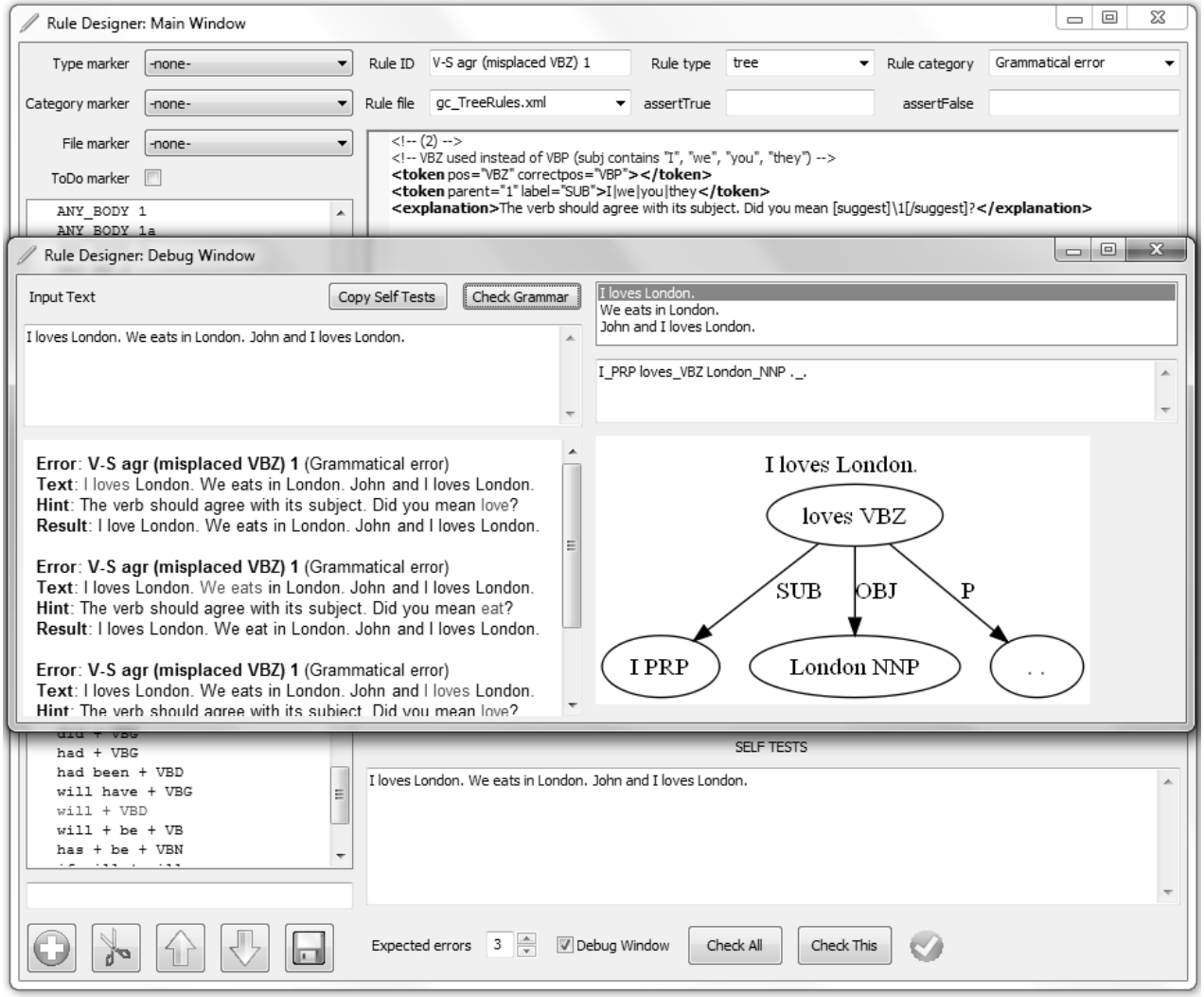

Figure 2: RuleDesigner.

with grammatically correct sentences. Thus, ungrammatical phrases may contain previously unseen patterns, causing incorrect results. For example, a part-of-speech tagger cannot reliably determine a tag for the word "like" in the phrase "he like dogs", since such a pattern never appears in the training collection.

Since processing ungrammatical sentences is a crucial feature for a grammar checking module, this issue needs further research. One of the possible solutions would be to extend the training collection with ungrammatical sentences. Our preliminary experiments have indeed shown that the inclusion of ungrammatical sentences into the training collection increases the quality of part-of-speech tagging.

7 Conclusion

We have designed and implemented the mechanism of natural language parser-backed rules for a LanguageTool-based grammar checking module. Our syntax allows designing the rules that analyze word-word dependencies in a given phrase. We have shown real examples of language phenomena, where such rules are much more helpful than built-in LanguageTool instruments.

Dependency-based rules are typically more complicated than the rules, based on regular expressions. Therefore, we developed RuleDesigner — a visual tool for rule authoring. It shows the user how language tools (sentence splitter, part-of-speech tagger, and dependency parser) process the input text, thus assisting debugging. We also included a system of self-tests, useful to keep the grammar checker consistent during the process of development.

References

[1] J. Burston (2008). Bon Patron: An Online Spelling, Grammar, and Expression Checker. *CALICO Journal*, vol. 25(2), pp. 337-347.

[2] H.J. Chen (2009). Evaluating Two Web-based Grammar Checkers-Microsoft ESL Assistant and NTNU Statistical Grammar Checker. *International Journal of Computational Linguistics & Chinese Language Processing*, vol. 14(2), pp. 161-180.

[3] B. O'Regan, A. Mompean and P. Desmet (2010). From Spell, Grammar and Style Checkers to Writing Aids for English and French as a Foreign Language: Challenges and Opportunities. *Revue francaise de linguistique appliquee*, vol. 15(2), pp. 67-84.

[4] E.M. Bender, D. Flickinger, S. Oepen, A. Walsh, and T. Baldwin (2004). Arboretum: Using a precision grammar for grammar checking in CALL. *Proceedings of the InSTIL/ICALL Symposium: NLP and Speech Technologies in Advanced Language Learning Systems*, Venice, Italy, pp. 83-86.

[5] M. Milkowski (2010). Developing an open-source, rule-based proofreading tool. *Software: Practice and Experience*, vol. 40(7), pp. 543-566.

[6] M.J. Alam, N. UzZaman and M. Khan (2006). N-gram based statistical grammar checker for Bangla and English. *Proceedings of ninth International Conference on Computer and Information Technology (ICCIT 2006)*, Dhaka, Bangladesh.

[7] J. Wagner, J. Foster and J. van Genabith (2006). Detecting grammatical errors using probabilistic parsing. *Workshop on Interfaces of Intelligent Computer-Assisted Language Learning*. Columbus, Ohio, USA.

[8] J. Sjobergh (2006). The Internet as a Normative Corpus: Grammar Checking with a Search Engine. *Technical Report,* KTH Nada, Sweden.

[9] D. Naber (2003). A rule-based style and grammar checker. *Master's thesis,* University of Bielefeld, Germany.

[10] A. Ratnaparkhi (1996). A maximum entropy model for part-of-speech tagging. *Proceedings of the conference on empirical methods in natural language processing*, Philadelphia, Pennsylvania, USA, pp. 133-142.

[11] J.C. Reynar and A. Ratnaparkhi (1997). A maximum entropy approach to identifying sentence boundaries. *Proceedings of the fifth conference on Applied natural language processing*, Washington D.C., USA, pp. 16-19.

[12] J. Nivre, J. Hall, J. Nilsson, A. Chanev, G. Eryigit, S. Kübler, S. Marinov, and E. Marsi (2007). MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, vol. 13(2), pp. 95-135.

[13] P. Jian and C. Zong (2009). Layer-Based Dependency Parsing. *Proceedings of the 23rd Pacific Asia Conference on Language, Information and Computation (PACLIC 23)*, Hong Kong, China, pp. 230-239.