Writing Support for Controlled Natural Languages

Tobias Kuhn
Department of Informatics
University of Zürich
Switzerland
`tkuhn@ifi.uzh.ch`

Rolf Schwitter
Centre for Language Technology
Macquarie University
Sydney NSW 2109, Australia
`rolfs@ics.mq.edu.au`

Abstract

In this paper we present interface techniques that support the writing process of machine-oriented controlled natural languages which are well-defined and tractable fragments of English that can be translated unambiguously into a formal target language. Since these languages have grammatical and lexical restrictions, it is important to provide a text editor that assists the writing process by using lookahead information derived from the grammar. We will discuss the requirements to such a lookahead text editor and introduce the semantic wiki AceWiki as an application where this technology plays an important role. We investigate two different approaches how lookahead information can be generated dynamically while a text is written and compare the runtimes and practicality of these approaches in detail.

1 Introduction

Natural language interfaces have been a popular area of research in the 70's and 80's, in particular natural language interfaces to databases received a lot of attention and some of these interfaces found their way into commercial applications (Copestake and Jones, 1990; Androutsopoulos et al., 1995). However, most of the early research prototypes focused on the exploration of specific NLP techniques and formalisms and were not very robust and therefore not very successful – and research drifted away from this topic in the 90's although there is still no agreed consensus theory on how to build these text-based interfaces.

The need for text-based natural language interfaces has recently gained again momentum since more and more non-specialists are required to access databases and maintain knowledge systems in Semantic Web applications through their web browsers and portable devices (Cimiano et al., 2008). One advantage of natural language interfaces is that the user is not required to learn a formal language to communicate with the system. But the use of natural language as an interface style is not without problems since full natural language is highly ambiguous and context-dependent. In practice, natural language interfaces can only understand a restricted subset of written (or spoken) input, and they have often only limited capabilities to make their coverage transparent to the user (Chintaphally et al., 2007). If the system is unable to understand a sentence and does not provide accurate feedback, then there is the risk that the user makes potentially incorrect assumptions about other sentences that the system actually can understand.

An obvious solution to this problem is to use a set of training sessions that teach the user what the system can and cannot understand. But this solution has at least two drawbacks: (a) untrained users cannot immediately use the system; and (b) infrequent users might not remember what they learned about the system's capabilities (Tennant et al., 1983a).

A better solution is to build natural language interfaces that directly support the interaction between the user and the system and that are able to enforce the restrictions of the language and guide the writing process in an unobtrusive way.

Menu-based natural language interfaces have been suggested to overcome many problems of conventional text-based natural language interfaces. These menu-based interfaces can be generated automatically from the description of a database or from a predefined ontology (Tennant et al., 1983b). These interfaces are very explicit about their coverage since they provide a set of (cascaded) menus that contain the admissible natural language expressions and the user has to select only from a set of choices. Therefore the failure rate is very low, in addition there are no spelling errors since no typing is required. Most menu-based and related WYSIWYM (= What You See Is What You Mean) approaches assume an existing underlying formal representation (Power et al., 1998). WYSIWYM is a natural language generation approach based on conceptual authoring where the system generates feedback from a formal representation. Instead of manipulating the formal representation, the user edits the feedback text via a set of well-defined editing operations (Hallett et al., 2007).

Although menu-based natural language interfaces can be applied to large domains, there exist applications where menu-searching becomes cumbersome (Hielkema et al., 2008) and more importantly there exist many applications where the formal representation does not exist in advance and where the construction of this formal representation is the actual task. This is the setting where controlled natural languages can make an important contribution. Machine-oriented controlled natural languages can be used as high-level interface languages for creating these formal representations, and we can try to support the writing process of these controlled natural languages in an optimal way.

In the following, we will illustrate how controlled natural languages can be applied in a semantic wiki context where a formal representation needs to be derived from a text that is human-readable as well as machine-processable, and then we will show how these techniques can be implemented in a logic programming framework.

2 Controlled Natural Languages (CNL)

Over the last decade or so, a number of machine-oriented controlled natural languages have been designed and used for specification purposes, knowledge acquisition and knowledge representation, and as interface languages to the Semantic Web – among them Attempto Controlled English (Fuchs et al., 1998; Fuchs et al., 2008), PENG Processable English (Schwitter, 2002; Schwitter et al., 2008), Common Logic Controlled English (Sowa, 2004), and Boeing's Computer-Processable Language (Clark et al., 2005; Clark et al., 2007). These machine-oriented controlled natural languages are engineered subsets of a natural language with explicit constraints on grammar, lexicon, and style. These constraints usually have the form of construction and interpretation rules and help to reduce both ambiguity and complexity of full natural language.

The PENG system, for example, provides text- and menu-based writing support that takes the burden of learning and remembering the constraints of the language from the user and generates a paraphrase that clarifies the interpretation for each sentence that the user enters. The text editor of the PENG system dynamically enforces the grammatical restrictions of the controlled natural language via lookahead information while a text is written. For each word form that the user enters, the user is given a list of choices of how to continue the current sentence. These choices are implemented as hypertext links as well as a cascade of menus that are incrementally updated. The syntactic restrictions ensure that the text follows the rules of the controlled natural language so that the text is syntactically correct and can be translated unambiguously into the formal target language (Schwitter et al., 2003; Schwitter et al., 2008).

3 Semantic Wikis

Semantic wikis combine the philosophy of wikis (i.e. quick and easy editing of textual content in a collaborative way over the Web) with the concepts and techniques of the Semantic Web (i.e. giving information well-defined meaning in order to enable computers and people to work in cooperation). The goal is to manage formal representations within a wiki environment.

There exist many different semantic wiki systems. Semantic MediaWiki (Krötzsch et al., 2007), IkeWiki (Schaffert, 2006), and OntoWiki

(Auer et al., 2006) belong to the most mature existing semantic wiki engines. Unfortunately, **none** of the existing semantic wikis supports expressive ontology languages in a general way. For example, none of them allows the users to define general concept inclusion axioms, e.g.:

- Every country that borders no sea is a landlocked country.

Furthermore, most of the existing semantic wikis fail to hide the technical aspects, are hard to understand for people who are not familiar with the technical terms, and do not provide any writing support.

4 AceWiki – a CNL-based Wiki

AceWiki (Kuhn, 2008a; Kuhn, 2008b) is a semantic wiki that tries to solve those problems by using the controlled natural language Attempto Controlled English (ACE) to represent the formal content.[1] The use of the language ACE allows us to provide a high degree of expressivity and to represent the formal content in a natural way. Easy creation and modification of the content is enabled by a special text editor that supports the writing process.

AceWiki is implemented in Java. Making use of the Echo Web Framework[2], it provides a rich AJAX-based web interface. Figure 1 shows a screenshot of an exemplary AceWiki instance containing information about a geographical domain. The ACE parser is used to translate the ACE sentences into first-order logic and, if possible, into the ontology language OWL (Motik et al., 2008). In the background, the OWL reasoner Pellet[3] is used to ensure that the ontology (consisting of the sentences that are OWL-compliant) is always consistent. The reasoner is also used to infer class memberships and hierarchies and to answer questions.

Following the wiki philosophy, the users should be able to change the content of the wiki quickly and easily. For this reason, it is important that the users are supported by an intelligent text editor that helps to construct valid sentences. AceWiki provides a predictive text editor that is
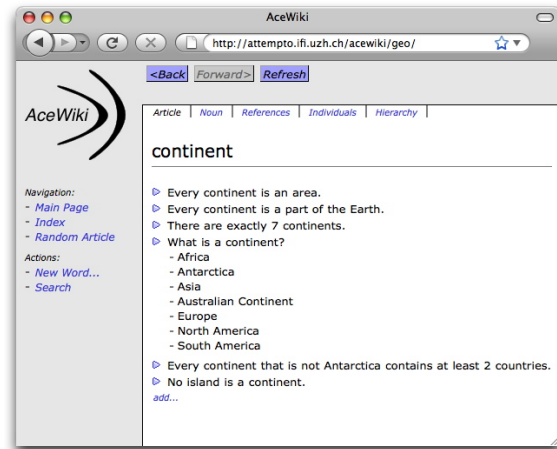

Figure 1: A screenshot of the web interface of AceWiki showing the wiki article for the class *continent*.

able to look ahead and to show possible words to continue the sentence. Figure 2 shows a screenshot of this editor. A chart parser is used for the generation of the lookahead information.

This predictive editor enables the creation of sentences by clicking consecutively on the desired words. It ensures that only 100% syntactically correct sentences are created. This enables novice users to create ACE sentences without the need to read the ACE manuals first. Alternatively, the predictive editor allows more experienced users to type a sentence or a part of it into a text field. The two modes of sentence construction (clicking versus typing) can be combined and the users can switch from one mode to the other at any time. Content words that are not yet known can be added on-the-fly when writing a sentence using the built-in lexical editor.

A usability experiment (Kuhn, 2008a) showed that people with no background in formal methods are able to work with AceWiki and its predictive editor. The participants — without receiving instruction on how to use the interface — were asked to add general and verifiable knowledge to AceWiki. About 80% of the resulting sentences were semantically correct and sensible statements (in respect of the real world). More than 60% of those correct sentences were complex in the sense that they contained an implication or a negation.

---

[1] http://attempto.ifi.uzh.ch/acewiki/
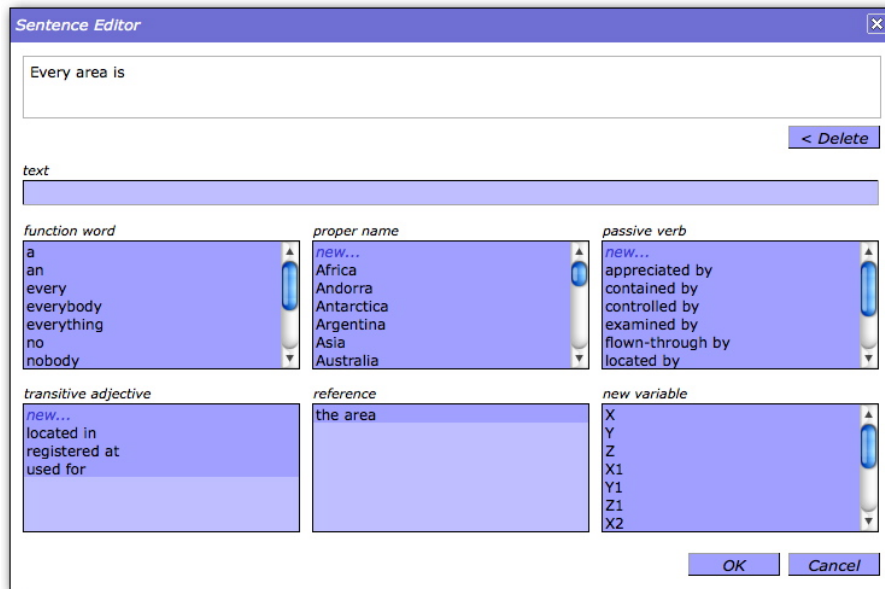[2] http://echo.nextapp.com/site/
[3] http://pellet.owldl.org/

Figure 2: A screenshot of the predictive editor of AceWiki. The partial sentence *Every area is* has already been entered and now the editor shows all possibilities to continue the sentence. The possible words are arranged by their type in different menu boxes.

5 AceWiki Grammar

AceWiki uses a subset of ACE. The grammar contains about 80 grammar rules and is implemented in a declarative way in Prolog. These rules are defined in a special grammar notation using feature structures, for example:

```
verbphrase(pl:PL,neg:Neg,that:T) =>
  verb(verb:full,pl:PL,neg:Neg),
  nounphrase(verb:full,that:T).
```

This format can be transformed into Java code (which is used by the AceWiki chart parser) and can also be exported as a Prolog DCG grammar (Pereira and Shieber, 1987).

The AceWiki language consists of about 30 function words (e.g. *a*, *every*, *if*, *then*, *and*, *not*, *is*, *that*) and five extensible types of content words (some of which have several word forms): proper names (e.g. *Europe*), nouns (e.g. *city*), transitive verbs (e.g. *contains*), *of*-constructions (e.g. *part of*), and transitive adjectives (e.g. *located in*).

The grammar covers a considerable part of English: singular and plural noun phrases, active and passive voice, negation, relative phrases, conjunctions/disjunctions (of sentences, verb phrases, and relative phrases), existential and universal quantifiers, questions, and anaphoric references. Below are a few examples of declarative sentences:

- Switzerland is located in Europe and borders exactly 5 countries.

- Every country that borders no sea is a landlocked country and every landlocked country is a country that borders no sea.

- If X borders Y then Y borders X.

The semantic expressivity of the AceWiki language is a subset of the expressivity of first-order logic. The mapping from ACE to OWL that is applied by the ACE parser covers all of OWL 2 except data properties and some very complex class descriptions (Kaljurand, 2007). Apart from those exceptions, the AceWiki language is more expressive than OWL. The examples shown above are OWL-compliant, but we can write sentences that go beyond the semantic expressivity of OWL, for example[4]:

- No country borders every country.

- If Berlin is a capital then Germany is not an unstable country.

- If a country contains an area and does not control the area then the country is an unstable country.

---

[4](Kaljurand, 2007) explains how OWL-compliant ACE sentences can be distinguished from ACE sentences that have no OWL representation.

At the moment, AceWiki supports only simple questions containing exactly one *wh*-word (i.e. *what*, *who*, or *which*). Such questions can be mapped to OWL class descriptions[5]. The answers to such questions can be calculated by determining the individuals of the class description. Here are three typical questions expressed in AceWiki:

- Which continents contain more than 10 countries?
- Switzerland borders which country that borders Spain?
- Which rivers flow through a country that borders Spain?

In summary, the AceWiki grammar describes a relatively large subset of English which exceeds the expressivity of OWL.

6 Implementing Writing Support

As we have seen, the AceWiki grammar is written in a format that can be read as a DCG and that is used by the AceWiki chart parser in order to generate lookahead information and logical formulas. In this section, we will focus on a specific aspect of the writing support and illustrate how syntactic lookahead information can be harvested from the DCG in a direct way and in an indirect way through a chart parser (as AceWiki does). In order to make a direct comparison of the two approaches, we use Prolog in both cases.

6.1 Harvesting the DCG

In order to harvest lookahead information directly from the DCG (for which Prolog supplies by default a top-down, left-to-right, backtrack parsing algorithm), we add a dummy token after each new word that the user enters and then parse the entire input string from the beginning. This looks like a very costly approach but the advantage of this approach is that it can be implemented very easily (and our empirical results show that this approach is still fast enough for practical applications). Furthermore, this approach prevents the parser from getting into infinite loops when processing recursive grammar rules. Let us assume that the user is planning to add the following sentence to the wiki:

- Switzerland is an alpine country.

---

[5]These queries are sometimes called "DL Queries".

and that she has just appended the indefinite article *an* to the string *Switzerland is*. Before this entire input string is parsed by the DCG parser, the dummy token '`$dummy$`' is added to the end of the string:

['Switzerland',is,an,'$dummy$']

This dummy token will be used – as we will see below – to trigger a Prolog rule (`word_form/4`) that processes the lookahead information. Before we describe this, we need to understand how preterminal grammar rules look like in the AceWiki grammar; here is a (simplified) example:

```
'NOUN' -->
  { lexicon(cat:'NOUN',wf:WF) },
  word_form('NOUN',WF).
```

The term in curly brackets is a Prolog expression. The default Prolog DCG preprocessor takes this DCG rule and translates it into a pure Prolog clause, adding two additional arguments for the input and output string. Note that the variable `WF` stands for an entire word form that can either be atomic or compound. This word form is represented as a list of tokens in the lexicon, for example (again simplified here):

```
lexicon(cat:'NOUN',
        wf:[alpine,country]).
```

This lexicon lookup (`lexicon/2`) returns the entire list of tokens and sends it to the rule `word_form/4` that processes these tokens depending on the input string:

```
word_form(Cat,[T1|Ts],['$dummy$'],_) :-
  update_lah_info(lookahead(Cat,T1)),
  fail.

word_form(Cat,[T1|Ts],[T1|S1],S2) :-
  word_form(Cat,Ts,S1,S2).
```

At this point, the remaining input string contains only the dummy token since all other words of this string have already been processed. This dummy token is processed by the first of the two grammar rules (`word_form/4`) that takes the first token (`T1`) of the compound word and adds it together with the category (`Cat`) to the lookahead information. Once the lookahead information is updated, the rule fails and additional lookahead information is collected via backtracking. The text editor will display the first token (`alpine`) of the compound word (`alpine country`) together with other lookahead information. If the

user selects `alpine` as the subsequent input, the dummy token is added again to the new input string but this time the second of the grammar rules (`word_form/4`) is triggered. This rule removes `alpine` from the input string before the dummy token is detected by the first rule that adds the next token of the compound (`country`) to the lookahead information. In order to avoid duplicates, the predicate `update_lah_info/1` first checks if the lookahead information already exists and only asserts this information if it is new:

```
update_lah_info(LAHInfo) :-
  (
    call(LAHInfo), !
  ;
    assert(LAHInfo)
  ).
```

Instead of atomic and compound word forms, we can also display syntactic categories for entire groups of words using the same technique.

6.2 Harvesting the Chart

As we have seen in the last section, the DCG is used to parse each sentence from scratch to process the lookahead information for a new word. Apart from that the DCG creates many partial structures and destroys them while backtracking. In order to avoid unnecessary repetition of work, the DCG can be processed with a (top-down) chart parser. A chart parser stores well-formed constituents and partial constituents in a table (= chart) consisting of a series of numbered vertices that are linked by edges (see Kay (1980) or Gazdar and Mellish (1989) for an introduction). Our chart parser is an extension of (Gazdar and Mellish, 1989) and represents edges in a similar way as a predicate with five arguments:

edge(V1,V2,LHS,RHSFound,RHSToFind)

The first two arguments state that there exists an edge between vertex `V1` and vertex `V2`. The next three arguments represent a grammar rule and indicate to what extent the parser was able to apply this grammar rule to the constituent found between the two vertices. Here `LHS` stands for a category on the left-hand side of a grammar rule, `RHSFound` for a sequence of categories that has been found on the right-hand side of the grammar rule, and `RHSToFind` for a sequence of remaining categories on the right-hand side. The edges in the chart are either *inactive* or *active* edges. An inactive edge is an edge where the list `RHSToFind` is empty (`[]`) and represents a confirmed hypothesis. All other edges are active and represent unconfirmed hypotheses.

The fundamental rule of chart parsing says that if an inactive edge meets an active edge of the corresponding category, then a new edge can be added to the chart that spans both the inactive and the active edges. In order to apply the fundamental rule, the chart needs to contain at least one active and one inactive edge. That means we have to initialise the chart first for each new sentence. This initialisation process will allow us to add a number of active edges to the chart and to extract the initial lookahead information from the chart. This information will show the user how to start a sentence. Once the user selects or types the first word, new inactive edges are triggered that are required by the fundamental rule to start the chart parsing process.

Before we can process the DCG with the chart parser, we transform it into a slightly different format using the infix operator `==>` instead of `-->`.

In the next step, we can initialise the chart top-down (`text/1`) and make sure that a number of active edges is generated. We do this with the help of a failure-driven loop (`foreach/2`):

```
chart_parser([],[_,0,_]) :-
  LHS = text([A,B,C,D,E,F,G,H,I]),
  init_chart(0,LHS).

init_chart(V0,LHS) :-
  foreach(rule(LHS,RHS),
    init_edge(V0,V0,LHS,[],RHS)).

init_edge(V0,V0,LHS,Fnd,RHS) :-
  edge(V0,V0,LHS,Fnd,RHS), !.

init_edge(V0,V0,LHS,Fnd,[RHS|RHSs]) :-
  assert_edge(V0,V0,LHS,Fnd,[RHS|RHSs]),
  foreach(rule(RHS,RHS2),
    init_edge(V0,V0,RHS,[],RHS2)),
  update_lah_info(RHS).

foreach(X,Y) :- call(X), once(Y), fail.
foreach(X,Y) :- true.
```

This failure-driven loop calls the transformed DCG rules via the following rules:

```
rule(LHS,RHS)  :- ( LHS ==> RHS ).
rule(LHS,[])   :- ( LHS ==> [] ).
```

and creates a set of active edges for the top-level grammar rules that start and end at vertex `0`. Additionally, this initialisation process asserts the

initial lookahead information `RHS` that is available as the first element in the lists of remaining categories `[RHS|RHSs]`. This is done with the help of the predicate `update_lah_info/1` that works in a similar way as the one introduced for the DCG in the last section.

Once the chart is initialised and the initial lookahead information is displayed, the user can select or enter the first word that falls under these classifications. This word is processed by the predicate `chart_parser/2`, starts at vertex `0` and generates in the case of an atomic word an inactive edge for each instance in the lexicon:

```
chart_parser(Word,[_,V1,V2]) :-
  start_chart(V1,V2,Word).

start_chart(V1,V2,Word) :-
  foreach(word(V1,Word,Cat),
    add_edge(V1,V2,Cat,[],[])).

word(_,Word,Cat) :-
  ( Cat ==> [ lexicon(cat:Cat,wf:Word),
              word(Word) ] ),
  call( lexicon(cat:Cat,wf:Word) ).
```

The chart parser adds these inactive edges to the chart using in a first step the second of the following rules (`add_edge/5`) and then applies the fundamental rule (`fund_rule/4`) recursively to inactive and active edges. Note that the first of the following rules checks for duplicates and the third rule applies first the fundamental rule, then predicts new active edges, and finally updates the lookahead information in the subsequent steps:

```
add_edge(V1,V2,LHS,Fnd,RHS) :-
  edge(V1,V2,LHS,Fnd,RHS), !.

add_edge(V1,V2,LHS,Fnd,[]) :-
  assert_edge(V1,V2,LHS,Fnd,[]),
  fund_rule(V1,V2,LHS,[]).

add_edge(V1,V2,LHS,Fnd,[RHS|RHSs]) :-
  assert_edge(V1,V2,LHS,Fnd,[RHS|RHSs]),
  fund_rule(V1,V2,LHS,[RHS|RHSs]),
  predict_active_edges(V2,RHS),
  update_lah_info(RHS).
```

If an inactive edge is asserted by the second rule, then the fundamental rule is applied to every active edge that can make use of this inactive edge:

```
fund_rule(V1,V2,RHS,[]) :-
  foreach(edge(V0,V1,LHS,Fnd,[RHS|RHSs]),
    add_edge(V0,V2,LHS,[RHS|Fnd],RHSs)).
```

If an active edge is asserted in the third rule, then the fundamental rule is first applied to every inactive edge that can make use of this active edge:

```
fund_rule(V1,V2,LHS,[RHS|RHSs]) :-
  foreach(edge(V2,V3,RHS,Fnd,[]),
    add_edge(V1,V3,LHS,[RHS|Fnd],RHSs)).
```

and then the predicate `predict_active_edges/2` is applied that looks for each grammar rule (`rule/2`) that has the category `RHS` on the left hand side (`LHS`) and creates new active edges for these rules starting at vertex `V2`:

```
predict_active_edges(V2,LHS) :-
  foreach(rule(LHS,NewRHS),
    add_edge(V2,V2,LHS,[],NewRHS)).
```

Finally, in a last step, the actual lookahead information (`RHS`) is updated with the help of the predicate `update_lah_info/1`.

Note that the rule `word/3` can only be used to process atomic words. We use additional rules to deal with compound words. The basic idea is to take a compound word out of the lexicon once the user enters the first token of a compound and write all remaining tokens onto a stack. The first token of these remaining tokens is then displayed as lookahead information and the stack is processed recursively as the user enters more tokens that belong to the compound word. This approach has the advantage that a compound word is looked up only once in the lexicon and the remaining tokens are processed directly from the stack.

7 Evaluation

We have compared the runtimes of the direct DCG approach with the indirect chart parsing approach for generating lookahead information. For the chart parser, we evaluated two settings: (a) the time that it takes to add a new token incrementally if the chart has already been constructed, and (b) the time that it takes to parse a partial sentence from the beginning. In the case of the DCG parser, those two settings coincide since incremental parsing is not possible.

For this comparison, we used the AceWiki grammar together with a test suite of 100 sentences. The longest sentence in this test suite consists of 51 tokens, the shortest sentence of 4 tokens, and the average sentence length is 14 tokens. Many of these sentences contain quite complex syntactic structures, among them embedded relative clauses, negation, coordination, and number restriction, for example:

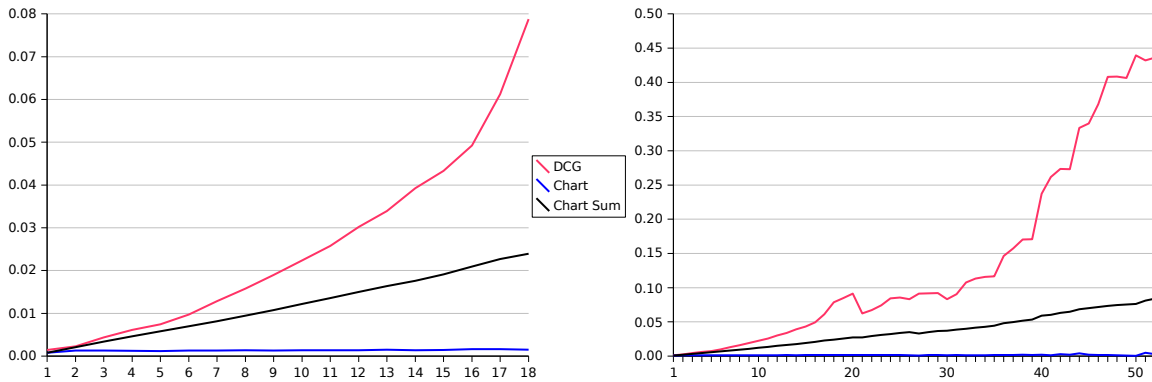- Sydney is a part of a country that is not a landlocked country and that borders at least two seas.

Figure 3: These two graphics show the average runtimes of the three parsing approaches in seconds (y-axis) for the different sentence positions (x-axis) up to position 18 (on the left) and 52 (on the right), respectively.

It turned out that the **average** processing times for generating lookahead information are: 1.3 ms for the chart parsing approach if only the last token has to be processed; 13.1 ms for the chart parsing approach if all of the partial sentence is parsed; and 31.3 ms for the DCG approach. These values where retrieved on a standard 2 GHz *Intel Core Duo* machine using SWI Prolog[6]. Not surprisingly, the latter two values highly depend on the number of tokens in a partial sentence. Figure 3 shows the average time values by sentence position (i.e. number of tokens) for the three approaches. We get very nice curves up to about 18 tokens, and then the DCG curve gets shaky. This is due to the fact that only 25% of the sentences have more than 18 tokens, and thus the sample gets more and more unreliable.

The chart parsing approach requires approximately constant time for each new token, and thus approximately linear time is required if the sentence has to be parsed from the beginning. In both settings, the chart parser performs better than the DCG parser. Surprisingly, not even the processing of the first token in a sentence is faster when we use the DCG instead of the chart parser (1.4 ms versus 0.8 ms). The theoretical worst-case time complexity of the DCG approach is cubic but on average a sentence of 15 tokens can be processed with the AceWiki grammar within 43.3 ms. That means that the DCG approach is still practically useful although theoretically not exhilarating.

8 Conclusions

In this paper we argued for controlled natural language (CNL) interfaces using predictive text editors to combine human-readability and usability with machine processability. As an example, we presented AceWiki, a semantic wiki, that uses a relatively complex grammar that is based on a subset of Attempto Controlled English (ACE) which is a machine-oriented CNL. A predictive text editor is used to enforce the restrictions of the CNL and to guide the user step-by-step via lookahead information that is generated on the fly while a sentence is written. We have explored two different approaches to generate this lookahead information: the first approach uses the DCG directly and is computationally costly but very easy to implement; the second approach relies on a chart parser and is computationally cheap but requires a bit more work to implement. We showed how the two approaches can be implemented in Prolog, and we compared those implementations. The chart parser performs particularly well if incremental parsing is applied. In any case, the chart parser performs considerably better than the DCG parser. However, the parsing times of the DCG approach are still within a reasonable range to be used in practical applications.

Providing adequate writing support is important for the acceptance of controlled natural languages. In the future, we will refine the presented techniques and study more sophisticated editing operations that can be applied to an existing text and require only minimal reprocessing.

References

I. Androutsopoulos, G. Ritchie, and P. Thanisch. 1995. Natural Language Interfaces to Databases – An Introduction. In: *Journal of Language Engineering*, 1(1), pp. 29–81.

S. Auer, S. Dietzold, and T. Riechert. 2006. OntoWiki – A Tool for Social, Semantic Collaboration. In: *Proceedings of the 5th International Semantic Web Conference*, pp. 736–749, Springer.

V. R. Chintaphally, K. Neumeier, J. McFarlane, J. Cothren, and C. W. Thompson. 2007. Extending a Natural Language Interface with Geospatial Queries. In: *IEEE Internet Computing*, pp. 82–85.

P. Cimiano, P. Haase, J. Heizmann, and M. Mantel. 2008. ORAKEL: A Portable Natural Language Interface to Knowledge Bases. In: *Data & Knowledge Engineering (DKE)* 65(2), pp. 325–354.

P. Clark, P. Harrison, T. Jenkins, T. Thompson, and R. Wojcik. 2005. Acquiring and Using World Knowledge Using a Restricted Subset of English. In: *Proceedings of FLAIRS'05*, pp. 506–511.

P. Clark, P. Harrison, J. Thompson, R. Wojcik, T. Jenkins, and D. Israel. 2007. Reading to Learn: An Investigation into Language Understanding. In: *Proceedings of AAAI 2007 Spring Symposium on Machine Reading*, pp. 29–35.

A. Copestake, K. Sparck Jones. 1990. Natural Language Interfaces to Databases. In: *Knowledge Engineering Review*, 5(4), pp. 225–249.

N. E. Fuchs, U. Schwertel, and R. Schwitter. 1998. Attempto Controlled English – Not Just Another Logic Specification Language. In: *Proceedings of LOPSTR'98*, pp. 1–20.

N. E. Fuchs, K. Kaljurand, T. Kuhn. 2008. Attempto Controlled English for Knowledge Representation. In: *Reasoning Web, Fourth International Summer School 2008*, LNCS 5224, pp. 104–124.

G. Gazdar, C. Mellish. 1998. *Natural Language Processing in Prolog*. An Introduction to Computational Linguistics, Addison-Wesley, 1989.

C. Hallett, R. Power, and D. Scott. 2007. Composing Questions through Conceptual Authoring. In: *Computational Linguistics*, 33(1), pp. 105–133.

F. Hielkema, C. Mellish, P. Edwards. 2008. Evaluating an Ontology-Driven WYSIWYM Interface. In: *Proceedings of INLG 2008*, pp. 138–146.

K. Kaljurand. 2007. *Attempto Controlled English as a Semantic Web Language*. PhD thesis, Faculty of Math. and Computer Science, University of Tartu.

M. Kay. 1980. Algorithm Schemata and Data Structures in Syntactic Processing. In: *CSL-80-12*, Xerox Parc, Palo Alto, California, 1980.

M. Krötzsch, D. Vrandecic, M. Völkel, H. Haller, and R. Studer. 2007. Semantic Wikipedia. In: *Journal of Web Semantics*, 5/2007, pp. 251–261, Elsevier.

T. Kuhn. 2008. AceWiki: A Natural and Expressive Semantic Wiki. In: *Proceedings of Semantic Web User Interaction at CHI 2008: Exploring HCI Challenges*. CEUR Workshop Proceedings.

T. Kuhn. 2008. AceWiki: Collaborative Ontology Management in Controlled Natural Language. In: *Proceedings of the 3rd Semantic Wiki Workshop*. CEUR Workshop Proceedings.

B. Motik, P. F. Patel-Schneider, I. Horrocks. 2008. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C Working Draft, 11 April 2008. `http://www.w3.org/TR/owl2-syntax/`

F. C. N. Pereira, S. M. Shieber. 1987. *Prolog and Natural-Language Analysis*. CSLI, Lecture Notes, Number 10.

R. Power, D. Scott, and R. Evans. 1998. What You See Is What You Meant: direct knowledge editing with natural language feedback. In: *Proceedings of ECAI 98*, Brighton, UK.

S. Schaffert. 2006. IkeWiki: A Semantic Wiki for Collaborative Knowledge Management. In: *Proceedings of the First International Workshop on Semantic Technologies in Collaborative Applications (STICA06)*.

R. Schwitter. 2002. English as a Formal Specification Language. In: *Proceedings of DEXA 2002*, September 2-6, Aix-en-Provence, France, pp. 228–232.

R. Schwitter, A. Ljungberg, and D. Hood. 2003. ECOLE – A Look-ahead Editor for a Controlled Language, In: *Proceedings of EAMT-CLAW03*, May 15-17, Dublin City University, Ireland, pp. 141–150.

R. Schwitter, M. Tilbrook. 2008. Meaningful Web Annotations for Humans and Machines using Controlled Natural Language. In: *Expert Systems*, 25(3), pp. 253–267.

J. F. Sowa. 2004. Common Logic Controlled English. *Draft*, 24 February 2004. `http://www.jfsowa.com/clce/specs.htm`

H. R. Tennant, K. M. Ross, R. M. Saenz, C. W. Thompson, and J. R. Miller. 1983. Menu-Based Natural Language Understanding. In: *Proceedings of the 21st Meeting of the Association for Computational Linguistics* (ACL), MIT Press, pp. 51–58.

R. H. Tennant, K. M. Ross, and C. W. Thompson. 1983. Usable Natural Language Interfaces Through Menu-Based Natural Language Understanding. In: *CHI'83: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 154–160, New York, NY.

C. Thompson, P. Pazandak, and H. Tennant. 2005. Talk to Your Semantic Web. In: *IEEE Internet Computing*, 9(6), pp. 75–78.