International Journal on Information and Control, vol. 3, 1999, ICI, Bucharest

Yet Another Head Driven Generator of Natural Language

Dan Tufiş

Abstract

The paper discusses some basic issues in natural language generation (NLG) and describes a head-driven NLG system for HPSG-like language descriptions. We address mainly the aspects of surface natural language generation starting from a meaning representation for the message meant to be verbalized. The representation of the linguistic knowledge (grammar and lexicon) as well as most important implementation issues (the generator is implemented in SMALLTALK) are described in details, commenting on the SMALTALK code.

1 Introduction

Several researchers have noted that NLG is a much less investigated area of natural language processing when compared with work on analysis, although many useful applications (machine translation, automatic summarization, question-answering, dialogue systems, CALL and Intelligent CALL, etc.) are supposed to take advantage of the developments in NLG (see for instance [Fawcet&Tucker,1989], [McDonald,1993], [Zock,1996], [Tufiş&Zock, 1997], [Dale&Mellish, 1998], etc). This could be explained partially because of a quite spread misconception that generation is just the reverse of analysis, but also due to the fact that ad-hoc generation methods (pattern-based, for instance) were found useful enough for the specific purposes of many applications. It was mainly the work in reversibility issues showing that although it is possible to use the same linguistic information for both analysis and generation, the processes themselves are quite different in terms of both algorithmic complexity and control and only the shallow phases of these two processes (parsing and surface realization respectively) could be considered more or less the reverse of one another. In the mid of '80s, there was carried successful research concerning unitary treatment of the parsing and surface generation leading to the unification of the knowledge representation for both of these [Shieber,1988], [Dymetman, Isabelle, 1988] and many others. The reversibility of the linguistic descriptions became possible by the adoption of declarative formalisms, mainly those based on unification of attribute-value structures.

The reusability of linguistic knowledge is a natural requirement due to the rapid

technological and conceptual advance seen in the last years and also, predictable for the future. If one considers the effort needed to implement a processing environment and the effort required for a wide coverage language description (according to some estimates this ratio is at least 1:100) the reusability criterion in encoding language descriptions should normally prevail efficiency. A linguistic description intimately related to a particular formalism, however performant it might be, is at risk of partial or even complete reformulating, if the programming environment has to be changed. This is a first aspect of the reusability. A second subtler one refers to avoiding fundamental decisions, in linguistic phenomena modeling, based upon a formalism or linguistic theory in fashion in a certain time. More difficult to be achieved, this aspect of the reusability may be provided by localizing very precisely all the elements specific to a certain formalism or linguistic theory.

The lexicalization trend in grammar modeling, that is nowadays a generalized practice, and which has been undeniably facilitated by the unification technology, is one possible answer to achieving reusability in the large. Probably this is one reason for their general acceptance. Our work is placed within this lexicalized, unification-based view of the language processing.

When speaking about generation, it is usually the case that the distinction made between what has to be said and how should it be said is further refined into smaller tasks, such as text planning, sentence planning and linguistic realization (Dale & Mellish, 1998). Coming closer to the implementation of a working architecture, Reiter and Dale (1997) identifies 6 main problematic areas, which although overlapping to some extent, define at a finer grained level the strategy-tactics dichotomy (cf. (Dale & Mellish, 1998)):

- Content determination - deciding what information should be included in the text, and what should be omitted;
- Document structuring – deciding how should text be organized and structured. It is often the case that the information representation out of which a generator is supposed to produce coherent text is not structured at a sentence equivalent granularity;
- Lexicalization (lexical choice) – choosing the particular words or phrases to best convey the intended meaning.
- Aggregation – somehow related to the problem of document structuring, this issue is concerned with folding (whenever the case) several information into one sentence realization form for the sake of fluency and stylistic quality;
- Referring expression generation – determining what properties of any entity should be used in referring to that entity (differentiating from potential similar referents)
- Surface realization – determining how the proper production of NL text will be done based on the information provided by previous 5 processing levels. This includes producing adequate morphological forms of the selected words and serializing them according to the syntactic restrictions specified by the grammar.

There is one pervasive problem, namely the nature and the format of the input a NL generator is supposed to produce texts from. This issue can be raised with respect to each of the 6 phases mentioned above and is known as the mapping problem or the problem of logical equivalence in computational linguistics [Shieber, 1993], [McDonald, 1993]. This

problem can be split into two: defining a loss-less mapping and defining a computationally efficient mapping.

We address here only the process of surface realization and assume an input representation consistent with the linguistic knowledge representation, avoiding therefore the mapping problem.

The structure of this paper is as follows: in the first part of the paper we present the general issues raised by natural language surface generation, show the limitations of the classical top-down and bottom up generation methods, and introduce the head-driven generation method. Then we discuss the linguistic representation of the input, the grammar and the lexicon and dwell on the implementation of our generator. The last section presents preliminary results, current limitations and future work.

2 Surface Generation of Natural Language

The declarative view of grammars as promoted by the unification formalisms, predicts that grammars themselves should be interpretable without regard to direction that they should be equally suitable for use by parsers and by generators. Put simply, in such an environment the generator works by taking a feature-structure (FS) and searching for grammar rules and lexical entries, which can relate subparts of the FS to words.

Among the different approaches to surface generation of natural languages the head-driven models appear to be the most successful. Our system is from this point of view, yet another head driven generator not committed to a specific linguistic formalism, requiring just a unification-based description of the linguistic knowledge which would unambiguously identify a logical form supposed to represent the semantics of the embedding feature structure. However, as the reader will notice, the grammar and lexicon descriptions are closed to an HPSG style description. We have selected this notation for the same reasons [Wilcock&Matsumoto, 1998] selected HPSG, namely that a head-driven generator is supposed to take full advantage of linguistic descriptions expressed in a head-based manner. As HPSG is head driven not only syntactically, but also semantically, it is natural to expect that a head-driven generator would be an appropriate design option. In [Wilcock&Matsumoto, 1998] there were shown some inherent difficulties when a head driven generator is supposed to work with the full HPSG textbook semantics [Pollard&Sag, 1994]. The difficulties arise due to the usual phrasal amalgamation of the quantifier storage and contextual background conditions[1]. With a lexical amalgamation of the quantifier storage as proposed by [Pollard & Yoo 1995] and of the CONTEXT feature as proposed by [Wilcock, 1997], the difficulties in using a head driven generator for an HPSG grammar are

[1] Wilcock (1997) shows that the Semantics Principle needs reformulation and it is replaced by three other principles: Semantic Head Inheritance Principle, Quantifier Inheritance Principle and Contextual Head Inheritance Principle. The rationale for this reformulation of the Semantics Principle is the reconciliation of the HPSG definition of semantic head (based on adjunct daughter or syntactic head daughter) and the one used by a head-driven algorithm (based on the identity of the logical forms). This reconciliation is achieved by including the unscoped quantifiers and background conditions in the logical form out of which a head-driven generator is assumed to produce the surface string.

(to a large extent, almost) removed. The remaining difficulties in Wilcock's generator are created by the limited power of implementation means (the template expansion and difference lists) offered by the programming environment and due to the lack of a dynamic constraint checking mechanism.

The surface (tactical [Thompson, 1977]) generator we will present here is based on an efficient head-driven algorithm similar to the one used in BUG1 (van Noord 1988]) but contrary to most other head driven generators this one is implemented in Smalltalk taking advantage of the language expressivity and computational properties. The generator is a mix variant of van Noord's BUG1 and Shieber's Head Driven Generation Algorithm [Shieber, 1988], which incorporates the advantages and remedy the deficiencies of both of them. The extensions proposed by [Wilcock&Matsumoto, 1998] have been also observed.

For efficiency improvement, several implementation issues have been considered (indexing the lexicon on the logical structures, parallel search, copying/destructive unification of feature structures, etc.).

In the following we will briefly discuss some computational problems inherent to simple control strategies (top-down or bottom-up) which motivated the mix strategy used in head driven generators (top-down and bottom-up). We will also shortly discuss the completeness and coherency [Wedekind,1988] and how they are achieved in our generator. Then, we will describe in some details the Smalltalk implementation of the generator.

Concerning the input to the surface generator we distinguish between the *content* specification, the one that would head-drive the generation and the *functional* specification the one that would goal-drive the restrictions application. The content specification will reflect in the lexical choices, while the functional specifications will manifest in the final realization of the selected lexical entries. For instance, a logical form such as *write (john, novel)* provide just the content specification and unless some functional *defaults* are assumed (cat: s, tense: present, number: singular, definiteness: indef, etc.) the logical form is grossly under-specified.

The approach we have taken here is to provide the surface generator with a feature-structure within which the logical form is just the value of a feature called *sem* which is supposed to be found among the first-level features of the feature structure. The functional specifications are supposed to be found embedded into the structure representing the value of the also first-level feature *syn*. The graphemic realizations of the lexical items (currently the lexicon is a word-form one, with all the values responsible for the inflected form being fully instantiated) are to be found in the first-level feature called *phon*. We assume that whatever information is provided on quantification (QSTORE) as well on presuppositions and other discourse information (CONTEXT) are embedded into the *sem* attribute structure. This would account for a lexical amalgamation of the QSTORE and CONTEXT sets as proposed by [Wilcock&Matsumoto, 1998].

While in the representation of the input for the surface generator described in this paper the syntactic and "semantic" information is mix-up, it is relatively simple to design an interface

separating the two types of information. In [Tufiş & all, 1994] is described such an interface called KRIL (Knowledge Representation to InterLingua) used to link the immersive language tutoring system FLUENT [Hamburger&all, 1993], with the natural language generator of the ATHENA multilingual natural language processing system [Felshin, 1993]. The structure in Figure 3a describes the input specification for the sentence "*Does every person love Mary?"* while the structure in Figure 3b corresponds to the sentence "*Every person loved her*".

```
(kril (to-say
      :action :love
      :agent(object :class :person
                    :attribute :every
                    :arole :quantity)
      :object1 Mary)
      (how-to
        :s-type :interrogative
        :tma'(:present)
        :how'((agent   :exophoric))
```

```
(kril (to-say
      :action :love
      :agent(object :class :person
                    :attribute :every
                    :arole :quantity)
                    :object1 Mary)
      (how-to
        :s-type :affirmative
        :tma'(:past)
        :how'((agent   :exophoric)
              (object1 :anaphoric)
```

Figure 3: Separating logical form from the realization specifications

This description is mapped (via lexicon inspection) into a more linguistically-oriented representation (called Interlingua) and then passed to the underlying surface generator.

3 Why Top-Down or Bottom-Up Generators Are Not Enough

It is known from the theory of parsing [Aho, 1972] that in case of top-down, left-right approaches, the grammar rules have to be free of left-recursion, otherwise the process will not end. Not only the direct recursiveness (such as VP→VP COMP) is forbidden, but also indirect ones (such as A→B C…Z; B→A D…Y). Put it more formally, if we denote by α a string of terminal grammar symbols, with Ω a string of either terminal or non-terminal grammar symbols and by FIRSTNT(X) ={Y1 Y2 … Yk} the list of non-terminals so that they are the left-most non-terminal of whatever sentential form $\alpha Y \Omega$ that can be generated by the grammar starting with X as a start symbol, than for the top-down generators we should have the relation $X \notin$ FIRSTNT(X). For a grammar involving feature-structure descriptions, the condition before should be modified by replacing the equality in the membership test by a unification condition, that is X should not unify with any Y in FIRSTNT(X).

The use of a rule such as VP→VP COMP in modeling natural languages is not an accident but a common construction in many languages for specifying the complements of a verb. The intent is that the rule should be applied recursively until the subcategorization list of the head verb is empty (saturated) or, put it otherwise, until all the complements of the verb will be consumed. In [Shieber&all, 1988] it is discussed an example for Dutch cross-serial verb constructions where subcategorization lists may be appended by syntactic rules resulting in indefinitely long lists. One could argue that syntactic information might block infinite recursion, as any *subcat* list has a fixed length (this length is specified in the lexical entries) but, this information is not available for top-down generators because they would look-up

the lexicon only in the final stage when preterminal categories are dealt with. A more detailed discussion and examples of how a simple top-down generator would fail with tiny left-recursive grammars are given in [van Noord, 1988] and in [Shieber &all, 1989].

Another kind of difficulty a simple top-down generator would face is discussed in [Shieber & all]. The problem is illustrated with a rule like: **s/S ➔ np/NP, vp(NP)/S**. Consider that the generation process is initiated with the goal *s/love(john, mary)*, which will fire the rule **s/S➔np/NP, vp(NP)/S**. If the generation proceeds with the leftmost non-terminal (the subgoal **np/NP**), the search space for the binding of NP will be huge (since any NP covered by the grammar would unify with the unrestricted semantics of **np**). Thus, in a large system, although not properly an infinite recursion, this kind of problem may cause the same hanging-around as a genuine left-recursion. And this, in spite of the **vp** semantics being bound (due to the variable sharing) to the semantics *love(john, mary)*. There are several solutions for this problem: using a kind of goal freezing as proposed by [Colmerauer, 1982], a static goal ordering as proposed by [Dymetman & Isabelle, 1988] or dynamic ordering (based on checking the instantiation of semantics of the candidate daughters) as proposed by [Wedekind, 1988].

The bottom-up generators do not have termination problems with the rule recursiveness since whatever needed lexical information is available from the beginning. Yet, they have other difficulties briefly mentioned below.

Most bottom-up generators based on Shieber's [Shieber, 1988] algorithm use a deduction controlled by an Earley-type chart structure. While this approach nicely works in parsing, with non-determinism limited by the data driven conduct of the algorithm, reversing it for generation (that is making it goal driven, applying the rules bottom-up and keeping the results in the chart) it is not straightforward and might be the source of high computational inefficiency.

Shieber solved this problem by imposing the grammar the semantic monotonicy condition. This restriction requires that the semantics of each daughter node of a rule should subsume some part of the semantics of the mother node of that rule. Apparently, this is a natural restriction in dealing with a compositional semantics approach (each sub-component of a phrase is supposed to contribute the semantics of the phrase), but on a closer look it reveals severe limitations with respect to the coverage of natural occurring texts: phrasal verbs (*call* some friends *up*), idioms (*kick the bucket*), expletive constructions (*it is* …), etc.

Also, the adaptation of the Earley type of deduction for generation purposes is not as efficient as in parsing, because the left-to-right scheduling of the algorithm (more oriented to the structure than to the meaning of the string) imposes some delayed decisions. A notorious example is the incertitude on some attribute values (such as case) for a NP preceding the head verbal phrase.

Most (surface) generators we are aware of construct explicitly or implicitly a syntactic tree (with specific syntactic or semantic restrictions) that is accepted by the underlying grammar, and according to which the syntactic validity of the output string is ensured. The way this

syntactic tree (whether virtual or real) is constructed, a traversal order is defined over the tree (the most usual, but not the only possible, are depth-first & left-to-right or breadth-first & left-to-right). However, as we have seen, for generation purposes a different traversal order would be needed: one that should be goal driven but which will consider the information in the lexicon whenever this would be necessary. Certainly, this is particularly relevant for lexicalized grammars.

4 Head-Driven Generation

A straightforward way to meet the requirements stated at the end of the last paragraph would be to use the information in the logical form of the generator's input in a top-down way of control, but to effectively produce the partial sub-strings (assembled in the final output) in a bottom-up manner, controlled by the information available in the dictionary. Put this way, it is obvious that the actual pieces of texts will be produced neither from left-to-right nor from right-to-left, but in a mix way.

The diagram below tries to suggest the head-driven approach in (surface) generation of natural language. So, providing the generator with a semantic representation of the intended output (LF) and a grammatical category to start, the process will go downwards until a lexical entry (corresponding to the head of the syntactic structure to be generated) is found to subsume the input information. In the diagram below, this lexical entry (a particular grammar rule with no right-hand side) is labeled as D.

Generator's input: category A and its logical form (LF)

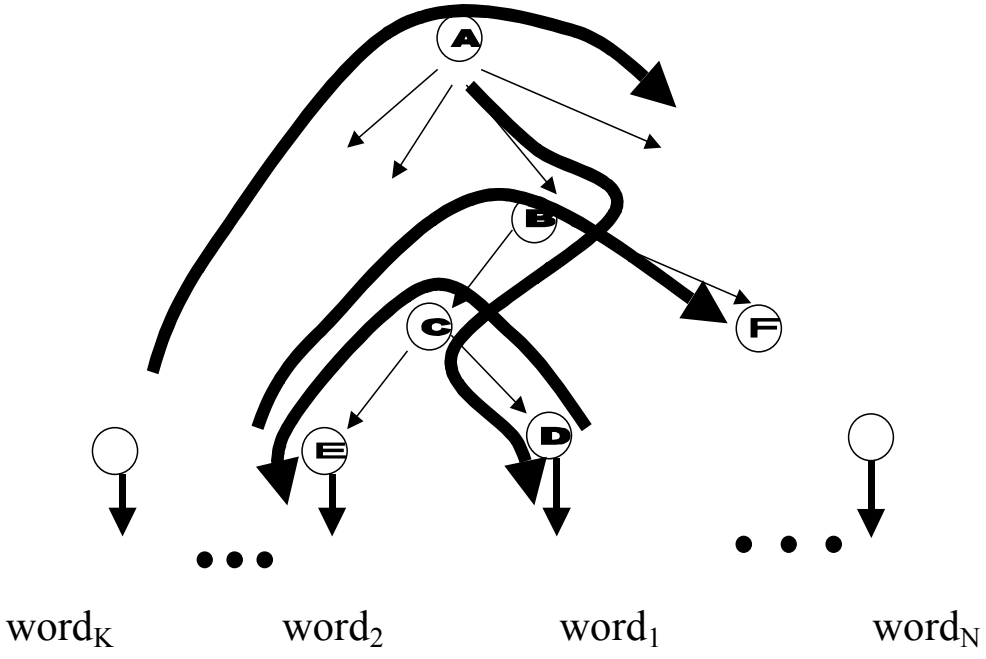


Figure 4 : The final N words text is not generated left to right

The result of unification (restrictions from the input LF and from the lexical entry) will be percolated upwards into the mother-node C. From here, C's non-head daughters are one by one submitted to the generation procedure. The generation of each daughter of C will contribute to further restricting the feature-structure of C (that is adding more information).

In the following, we will exemplify our implementation with the simplified version of the SMALLTALK code and therefore, the reader is assumed to have some general knowledge about object programming paradigm.

5 The dictionary and the grammar

The generator does not commit itself to a particular linguistic theory, but relies on a unification-based formalism. However, the lexical entries and the grammar rules exemplifying the functioning of the surface generator are encoded using an HPSG-flavored notation, but again we emphasize the idea that the basic functionality of the generator is not dependent on this formalism. The indexes showing the structure-sharing information are represented in the notation `@n:` followed by a feature structure FS and as `@n` for any reference to FS. A feature structure may contain only references (@n) which mean that all co-indexed references refer to the same unbound feature structure.

The top level structure of a lexical entry is shown in Figure 5

$$\begin{bmatrix} \text{phon} & \text{< a string >} \\ \text{syn} & \text{< a feature structure >} \\ \text{sem} & \text{< a sem - feature structure >} \mid \text{< an atom >} \end{bmatrix}$$

Figure 5: The top-level of a lexical entry

The features *phon* and *reln* take atomic values (strings of characters, representing the graphemic form of the word encoded by the entry and the name of the semantic predicate underlying the basic meaning of the word in case. The feature *syn* is non-atomic, having as values feature structures. In principle, *sem* takes non-atomic values, but in this current version we allowed also atomic values in order to let the generator work also with simpler grammars and lexicons. Given that the *sem* feature is obligatory for all words in the lexicon, usually atomic *sem* values are associated with description of functional words and constants (such as proper names).

The general structure of a *sem* feature structure is as follows:

reln	semantic_predicate
< argument1 - name >	< argument1 - value >
…	
< argumentn - name >	< argumentn - value >
qstore	< list >
conx	< list >

Figure 6: The general structure of a *sem* feature-structure

For instance, the semantics of the words in the paradigmatic family of *hate* is represented by the following structure: `(reln hate hater @1 hated @2)`

For building lexical entries, the system object `Lex` must be sent the method `withValues` with a `FeatureStructure` object as parameter as in the example below.

```
A := Lex withValues #(phon beautiful
                      syn (head (mod agr @1 def indef )
                            subcat (first (syn (head (n agr  @1 def indef) )
                                        sem @2)
                                  rest NIL))
                      sem (reln beautiful arg @1)).
```

The next examples, show the skeletons on the lexical entries for the words loves, I and Julie, based on which the generator, with an appropriate grammar, is able to produce the strings: “I love Julie” or “Julie falls in love” from logical the structures love(i, julie) and love(julie) respectively.

```
(phon loves
 syn (head (v vform (finite tns prs)
      subcat (first (syn (head (n agr (per a3 num sg)))
                     sem @1)
                rest (first (syn (head (np)) sem @2)
                      rest NIL)))
 sem (reln love lover @1 lovee @2)).

(phon “falls in love”
  syn (head (v vform (finite tns prs))
       subcat (first (syn (head (np agr (per a3 num sg)))
                      sem @1)
                rest NIL))
  sem (reln love lover @1)).

(phon I  syn (head (n agr (per a1 num sg)))  sem i)).

(phon Julie  syn (head (n agr (per a3 num sg))) sem julie)).
```

The grammar is an ordered set of rules (OrderedCollection). A rule is a feature structure pretty much the same as a lexical entry, having an additional (built-in) attribute called *dtrs* (daughters) which on its turn contains three (built-in) attributes: *l_dtrs* (left daughters), *head_dtr* (head daughter) and *r_dtrs* (right daugthers). *l_dtrs* and *r_dtrs* have as values ordered lists of features structures (they may be empty) corresponding to the daughters of the rule that precede and respectively follow the head daughter. If the value of a list of

daughters is the empty list, then the corresponding attribute is, as usually, omitted. The *head_dtr* has as value a feature structure, describing the head daughter of the rule, that is the one that shares the **sem** value with the mother node.

Two rules, corresponding to the more familiar format S→X VP and respectively VP→VP X^+, are shown in the figure 7 below (the notation {A . B} represents a list the first element of which is A and the rest is B):

```
R1:  (syn (head (s))
      sem @1
      dtrs ( l_dtrs @2
            head_dtr (syn (head (vp vform (finite)) subcat @2)
                        sem @1))).
R2:  (syn  (head @1 subcat  @2)
      sem  @3
      dtrs  (head_dtr (syn (head  @1:(vp vform (finite)) subcat {@2 . @4}) sem @3)
            r_dtrs @4))).
```

Figure 7: Two grammar rules

A new rule is added to the grammar by sending the system object `Rule` the method `withValues` with a `FeatureStructure` object as parameter as in the example below.

```
A := Rule withValues:  #(syn (head (np def indef @1 nil))
                           sem @2
                           dtrs(r_dtrs @3:(first(syn(head(np def indef agr @1))
                                                  sem @4)
                                           rest NIL)
                                head_dtr (phon @5
                                          syn (head (mod)
                                                subcat (first @3   rest NIL))
                                          sem (reln @5 arg  @4)))).
```

It is important for the generator to be able to distinguish efficiently between different words in the lexicon and to find all the rules that are applicable in a specific point of the generation. Given that the generator is driven by the semantics of the generation goal as well as by the lexical items that will be inserted in the output string, the dictionary (object of the type OrderedCollection) was indexed over the *sem* attribute. The dictionary accessor, `withSemantics: aSem`, accepts as search key (aSem) either an atomic value (as *i* and *julie* in the previous examples) or a proper *sem* structure (containing the *rel* attribute as in the example below:

```
(sem (reln love lover @1 lovee @2)).
```

The lexicon semantic index is a hash code table, which when given a *sem* value would return all the lexical entries whose semantics *strongly unifies* with it. *Strong unification* as implemented in our generator, requires that two unifiable feature structures have the same features the values of which should be pairwise unifiable. Therefore, under *strong unification* the feature structure (Feature1 x Feature2 y) would not unify with (Feature1 x). From this point of view, the *sem* feature structure is interpreted as a predicate and therefore two *sem* structure with the same *reln* value but with different number of features would be considered distinct and never unifiable. Except for the lexical access, the generator uses the

usual (and more permisive) unification which assumes only non-conflicting values for the same atttributes.

The grammar is searched for applicable rules on the basis of a head feature structure. The grammar accessor, `withHead: aHead`, returns all the rules the head of which unifies with the key (`aHead`)..

In the current version of the system, the lexicon contains all the inflectional variants of a given lemma. Therefore, the value of a key in the semantic index is usually a list of feature structures corresponding to the members of the inflectional family of the lemma or its synonyms. The list of lexical entries returned by the semantic index will be filtered out during the generation process due to unification failures (for instance a plural form will be filtered out by a unification with a feature-structure waiting for a singular form). However, with synonyms this syntactic filtering won't work and here is one point where an additional information source must be used to make a choice.

6 The generator

The core of the generator is implemented by the methods **generate, generateDaughters** and **upTowards:havingGenerated:**, which function in a co-routine regime.

To keep track of the potential multiple solution in the generation process, we implemented a specialized object called *StringVariants* (which is a type of chart, containing all possible sub-strings generated towards the final solution) and an appropriate method (called **variance**) which adds new sub-strings and combines (adding to the left and to the right) partial chunks of text into larger ones.

The generation procedure is initiated by looking up the lexicon for the entries that could serve as *generation pivots* for the input goal, that is whose *sem* values unify with the semantics of the current goal. In case the lexicon contains several such entries, then all of them will be selected for further processing (given that the words in the lexicon are indexed over their *sem* feature, all the potential pivots are collected within one lexicon look-up).

The unification underlying this implementation is a destructive one, i.e structure modifying. However, in order to avoid permanent data modification (grammar and lexicon) the original data is copied before using it. This way, the algorithm has been accelerated to a significant extent since a lot of frequent tests and auxiliary data structures necessary to keep the language resources unmodified, became unnecessary. As a side effect, this method eliminated the potential incompleteness of the generation algorithm (we will address this issue later). A successful unification operation modifies the arguments as a side-effect. The return value is a *block of assignments* which when executed will recreate the original arguments. This value is useful for undoing some modifications that led into a dead-end and resume the computation from other possible decision points.

```
generate
    | pivots result self_copy |
    result := StringVariants new.
```

```
   pivots := self lexicon withSemantics: (self at: #sem).
   pivots do: [ :p |
      self_copy := self copy.
      result addAll:
            (p upTowards: self_copy havingGenerated:
              (StringVariants with: ((p at: #phon) value asString)))].
   ^result
```

With a copy of the initial goal feature structure and each of the potential goals, the generator proceeds bottom-up (the method `upTowards:havingGenerated:`) looking for appropriate rules that will be applicable for the selected pivot. As in Shiber's and van Noord's generators the pivot plays the role of the semantic head of the grammar rules, that is that daughter of a local tree (representing one grammar rule) whose semantics is identical (structure sharing) with the semantics of the top of the tree (the mother node, or the left part of the rule).

The potential multiple generated texts are collected and appropriately stored into the *result* object which is returned as the output of the *generate* method.

```
upTowards: top havingGenerated: someStringVariants
   | rules result left right all |
   (self subsumed: top) isNil
     ifFalse: [^someStringVariants]
     ifTrue:
          [result := StringVariants new.
       rules := self grammar withHead: self.
       rules do:
         [:r | left := r lefts isNil
                 ifTrue: [StringVariants with: '' ]
                      ifFalse: [r lefts generateDaughters].
                right := r rights isNil
                 ifTrue: [StringVariants with: '' ]
                 ifFalse: [r rights generateDaughters].
         all := (left variance: someStringVariants) variance: right.
                    result addAll: (r mother upTowards: top havingGenerated: all)
          ]  ].
   ^result.
```

```
GenerateDaughters
    "generates the text corresponding to a list of daughters"
         self value == #NIL
   ifTrue: [^StringVariants with: ' ']
   ifFalse: [^(((self at: #first) generate)
                variance:
                ((self at: #rest) generateDaughters))].
```

When the top goal subsumes the input pivot - `(self subsumed: top)` -, the recursion in generation is stopped (the recursion is achieved via `generateDaughters` method). This condition test reaching at the top of the generation tree (obtaining a string permitted by the grammar), and also checks whether all the information provided in the initial goal has been used. The method simply returns the text that has been assembled by previous computations.

The indexing over the grammar rules allows for identifying the rules having a head daughter that unifies with the input pivot. The mechanism used in our implementation is similar to the *link* (precompiled) predicate used in van Noord's BUG1 generator.

So, if the top goal has not been reached, the relevant rules for the current goal are collected and each of them (`rules := self grammar withHead: self.`) is recursively applied for

generating an alternative piece of text (`rules do:[:r |…]`). With the text corresponding to the head node generated in the previous step, the **generate** method is sent to each of the left daughters (processed in the order given by the precedence order specified in the rule) getting the text(s) that are to be inserted to the left of the text(s) generated for the head node (`left := r lefts isNil ifTrue:[StringVariants with: ''] ifFalse: [r lefts generateDaughters]`). The same way, the generator computes the text(s) to be inserted to the right of the generated(s) from the head node (`right := r rights isNil ifTrue:[StringVariants with: ''] ifFalse:[r rights generateDaughters]`). Finally, the texts corresponding to the left daughters, the head daughter and the right daughter are assembled together (`all := (left variance: someStringVariants) variance: right`) and the mother of the currently processed rule is set as the new pivot in the bottom-up proceed of the algorithm (`r mother upTowards: top havingGenerated: all`). The methods **left**, **right** and **mother** are selectors which when sent to a rule, return the list of feature-structures for the left daughters, the list of feature-structures for the right daughters and respectively the feature-structure of the mother node pertaining to the rule in case. If the rule has no left or right daughters, the corresponding piece of text would be the empty string.

As any head-driven algorithm, the present one has no problem with the left recursiveness of the grammar rules. Given that this is argued at length elsewhere [vanNoord,1988], [Shieber, 1988] we won't get into further argumentation.

For building generation goals, one should send the system object `Generator` the constructor method `withValues` with an appropriate feature structure (that is one containing at least the *sem* atribute).

The basic assumption here is that whatever goal is given as input to the generator, there will be at least one lexical entry the semantics of which would unify with the semantics of the input. Otherwise, the generation procedure will produce an empty string. Apparently, this condition creates the same problems as Shieber's grammar monotonicity requirement, but in fact this is not so. The insertion of words stipulated either lexically (as in the case of particles or idioms) or syntactically (for instance, the expletive expressions) is always possible given that the stipulation is done in terms of **sem** restrictions. For instance, the lexical entry below[2], would be selected when the generator will be invoked with a goal such as *die(john)* and the idiom *kick the bucket* will be eventually generated. The non-idiomatic *kick* will not be consider since its semantics (`reln kick kicker @1 kickee @2`) does not unify with the initial goal.

```
(phon kicked
```

[2] A much simpler solution for generating the idiomatic expression would have been to use a simplified entry as the one given below but we wanted to exemplify the proper lexical insertion, that is one that would pick up whatever stipulated insertion from the lexicon:

```
(phon kicked_the_bucket
 syn (head (vp vform (finite tns past)
      subcat (first (syn (head (n agr (per a3 num sg))) sem @1)
                rest NIL))
 sem (reln die defunct @1)).
```

```
syn (head (vp vform (finite tns past)
     subcat (first (syn (head (n agr (per a3 num sg))) sem @1)
                   rest (first (syn (head (n def definite agr (per a3 num sg)))
                                    sem bucket))
                         rest NIL))))
sem (reln die defunct @1)).
```

Thus the sequence below, will yield two sentence-strings licensed by the grammar and lexicon:

```
A := Generator withValues: #(syn (head (s)
                                 sem (reln die
                                      defunct john)).
A generate. → StringVariants('John died' 'John kicked the bucket')
```

Among the problematic issues connected to the head-driven generation algorithm, both van Noord and Shieber discuss Wedekind's notions of coherence and completeness of a generator [Wedekind (1988)]. In its original form, Shieber's algorithm is both incoherence (that is overgenerating for underspecified input -unbound variables- with a non-terminating threat) nor complete (that is, undergenerating by leaving out some arguments of the input logical). The later version of Shieber's generator adopted van Noord's solution from BUG1: using *numbervars* to *freeze* the unbound variables in the input logical form (for ensuring coherency) and checking the subsumption of the final generated feature structure(s) by the initial input goal (for ensuring completeness). The specific solutions adopted in BUG1 were imposed mainly by the implementation of the underlying PROLOG unification.

Underspecification of *sem* structures do not create special problems in our generator because as we said before the *sem* structures are interpreted in a predicate-like manner. However, one could produce an underspecified *sem* structure which will not be filtered out by the lexical look-up, namely proving a nil (unbound) value for one or more features:

a) *sem (reln love lover john lovee nil)*

As these features are supposed to be referenced in the *syn* structure of the selected entries, the applicable grammar rules will eventually need these unbound features and they will fail and in the end the whole generation will fail. Such will always produce an empty string.

There might be, however, a real need to underspecify a certain feature. For such a case the generator allows a "don't care" value for any attribute (_) which is recorded in the lexicon and has the empty string as the value for the *pron* feature. Therefore, if the intended text is *"John loves", then the proper *sem* structure of the generation top goal should be in *sem (reln love lover john lovee _).*

Let us assume that the dictionary contains the following entries:

```
Entry1:
(phon loves
 syn (head (v vform (finite tns prs)
      subcat (first (syn (head (n agr (per a3 num sg)))
                     sem @1)
               rest (first (syn (head (np)) sem @2)
                     rest NIL)))
 sem (reln love lover @1 lovee @2)).

Entry2:
(phon falls_in_love
  syn (head (vp vform (finite tns prs))
       subcat (first (syn (head (np agr (per a3 num sg)))
                      sem @1)
                rest NIL))
  sem (reln love lover @1)).
```

With these lexical entries, asking the generator to produce a text from the following semantic specification (*sem love lover john)* only Entrie2 will be retrieved from the lexicon (due to the strong unification used by the lexical look-up). Based on it, the generator will produce the string "John falls in love".

On the other hand, if the generation will start with the semantic specification (*sem love lover john lovee _)* the actual generator will produce not the acceptable sentence "John falls in love" (as one might expect) but the questionable one *"John loves".

Completeness is the property of the algorithm ensuring that all information specified in the input is included in the generated output, that is when fed with a semantic representation like *sem (reln eat eater John food pie)* the generator should not provide as a result *John eats,* but *John eats a pie*. This property of the algorithm is achieved by the same copying and final subsumption mechanism as in BUG1

The proper generation is launched by constructing a generation object (by sending the system object `Generator` the constructor method `withValues with a FeatureStructure` object as parameter) and sending to it the method `generate`.

```
Generator withValues: #(syn (head (s))
                         sem (reln love
                              lover i
                              lovee julie)) generate.
➔I love Julie.

Generator withValues: #(syn (head (s))
                         sem (reln love
                              lover julie
                              lovee NIL)) generate.

➔Julie falls in love.
```

7 Conclusions and further work

The generator presented so far has been tested on a very small grammar and lexicon. More realistic language resources are needed for a thorough evaluation of the generator performance (both linguistic and computational). We have not discussed the strategic aspects of generation, that is how and from what a ***sem*** feature structure as required by the generator is constructed. Also, we have not discussed generation of multi-sentence natural text, planning the text structuring. A general solution to the strategic generation (if possible) requires further investigation, but it is very likely that a useful (especially from the computational performance point of view) should consider the client application of the surface generator, at least its knowledge representation and domain specific knowledge.

Aknowledgements

The work reported here was supported by a NATO grant at LIMSI/CNRS Orsay. It was part of a common project carried together with Michael Zock and many of his stimulating ideas were included into this surface generator. Mihai Ciocoiu wrote an initial Smalltalk version of the generator and most of the test grammar and dictionary.

Joseph Mariani and Gerard Sabah deserve special mention for making my stay at LIMSI extremely pleasant and stimulative.

8 References

Aho, A.V., Ulman, J.D. (1972) "The Theory of Parsing, Translation and Compiling." Prentice - Hall Series in Automatic Computation, 1972.

Colmerauer, A. (1983) PROLOG II: Manuel de réference et modèle théoriue. Technical report, Group d'Intelligence Artificielle, Marseille, France

Mc Donald, D. (1983) "Description Directed Control: Its Implications for Natural Language Generation." In Comp. & Mat. with Appls. Vol. 9 No. 1, 1983, pp. 111-129

Mc Donald, D. (1993) "Issues in the Choice of a Source for Natural Language Generation" In Computational Linguistics Volume 19, No. 1 pp. 191-197

Dale, R., Melish, C. (1998). – "Towards Evaluation in Natural Language Generation." In Proceedings of the 1st International Conference on Language Resources and Evaluation, Granada, Spain, pp. 555-562.

Dymetman, M, Isabelle, P (1988). - "Reversible Logic Grammars for Machine Translation". In Proceedings of the 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages, Pittsburgh.

Fawcet, R.,P., Tucker, G., H. (1989) – "Prototype Generators 1&2", Report No. 10 of COMMUNAL, Computational Linguistic Unit-University of Wales College of Cardiff, 30p.

Felshin, S. (1993) – "A Guide to the Athena Language Learning Project Natural Language Processing System". Copyright, Massachusetts Institute of Technology.

Hamburger, H., Tufiş, D. and Hashim, R. (1993) – "Structuring two-medium dialog for learning language and other things". In Proceedings of the Association for Computational Linguistics Workshop on Intentionality and Structure in Discourse Relations. Columbus, Ohio, June 20.

Shieber, S., van Noord, G., Moore, R. C., Pereira, F.C.N. (1989). - "A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms", in Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics, Vancouver.

Shieber,S.M.(1986). - "An Introduction to Unification-Based Approaches to Grammar", Lecture Notes CSLI, No. 4.

Shieber, S.M. (1988). - "A Uniform Architecture for Parsing and Generation". In Proceedings of the 12th International Conference on Computational Linguistics, Budapest, Hungary.

Shieber, S. (1993) "The Problem of Logical Equivalence in Computational Linguistics Volume 19, No. 1 pp. 179-190

Thompson, H. (1977). – "Strategy and tactics: A model for language production." In Papers from the 13th Regional Meeting, Chicago Linguistic Society, Chicago, Illinois, 651-668

Tufiş, D. Hamburger, H., Hashim, R., Pan, J. (1994) "Generation of Natural Language in an Immersive Language Learning System", in M.Browse (ed.) Basics of Man-Machine Communication for the Design of Educational Systems, Springer Verlag, 1994

Tufiş, D., Zock, M. (1997) – "Maximal Cohesion, Minimal Commitment and Lexical Pattern Matching", LIMSI Research Report, December, 1997, 56p.

Zock, M. (1996). – "The Power of Words in Message Planning." In *Proceedings of COLING'96*, Copenhagen Denmark, pp. 990-995