FROM WATER TO WINE: GENERATING NATURAL LANGUAGE TEXT FROM TODAY'S APPLICATIONS PROGRAMS [1]

David D. McDonald
2 Harrison Avenue
Northampton, MA 01060
MCDONALD@CS.UMASS.EDU

Marie W. Meteer
BBN Laboratories
10 Moulton St.
Cambridge MA 02338
MMETEER@G.BBN.COM

ABSTRACT

In this paper we present a means of compensating for the semantic deficits of linguistically naive underlying application programs without compromising principled grammatical treatments in natural language generation. We present a method for building an interface from today's underlying application programs to the linguistic realization component Mumble-86. The goal of the paper is not to discuss how Mumble works, but to describe how one exploits its capabilities. We provide examples from current generation projects using Mumble as their linguistic component.

INTRODUCTION

Work in artificial intelligence has two goals: on the one hand to do concrete work that can be used in actual systems today, on the other to establish strong theoretical foundations that will allow us to build more sophisticated systems tomorrow. Unfortunately, since the field is so young and so few problems are well understood, these two goals are often at odds.

Natural language generation is no exception. The years of research in linguistics have made problems in syntax comparatively well understood. Nevertheless, we should not restrict ourselves to just generating single, isolated sentences until the problems of lexical semantics, discourse structure, and conceptual modeling are understood as well. We must find ways to facilitate both efforts, modularizing our systems so that the parts that handle well understood processes need not be compromised to accomodate weaknesses in other parts of the system. This paper is on how to support such modularity in a natural language generator.

In the present case, the well understood process is linguistic realization, and the weaknesses are in the conceptual models and representations of the programs underlying the generator. To bridge this gap, we present a **specification language**, to be used as input to the linguistic realization component Mumble-86.[2] This language provides the designer of a planning component with a vocabulary of linguistic resources (i.e. words, phrases, syntactic constructions) and a straightforward means of directing their composition. The specification language facilitates interfacing Mumble to a wide range of underlying programs and planners. For simple programs not built with language in mind, we show a straightforward means of using predefined templates to map underlying objects to complex linguistic structures. For systems with more sophistication in text planning, we show how the compositionality and flexibility of the specification language can be used to make their task easier. What is template driven at one end of the range can be built compositionally at the other; what is stipulated at one end can be reasoned about at the other.

A MOTIVATING EXAMPLE

Consider the description *"53rd Mechanized Division"*. In most programs today a sufficient representation of the object it names could be just the symbol 53RD-MECHANIZED-DIVISION. The print name of the symbol conveys all the information that a person reading the code needs to know, without it actually playing a role in the program's reasoning. If all we cared about were a single communicative context, we might consider implementing the link between the symbol and the description as though the phrase were one long word without any internal structure. This expedient treatment would severely

[1] This work was supported in part by DARPA contracts N00014-87-K0238 at the Univerisity of Massachusetts and DAAA 15-87-C0006 CDRLA002 at BBN Laboratories, and by the Rome Air Development Center contract number AF30602-81-C-0169, task number I74398 at the University of Massachusetts.

[2] For a comprehensive description of Mumble-86, see Meteer, McDonald, Anderson, Forster, Gay, Huettner, & Sibun 1987.

limit our options, however. Indefinite references, such as "*a mechanized division*", and subsequent references, "*the division*", would have to be handled separately. Pronominalization would not be possible since there are no associated features such as number, gender, and person. Furthermore, since an artificial word would have no internal syntactic structure, a speech production program would have no information on which to base intonation. A better treatment is to introduce into the interface itself some of the generality and structure that the underlying representation is missing.

In the ALBM interface being developed at BBN, we associate an object like 53RD-MECHANIZED-DIVISION with the application of a general template to an explicit set of arguments as shown below:

```
(define-default-specification
                 '53rd-mechanized-division
   :template-name armed-forces-unit-name
   :arguments ("53rd"  "Mechanized"  "Division") )
```

FIGURE 1

By going to this slightly greater effort, we have supplied a hook for handling subsequent reference or other abstractions ("*the 53rd and 42nd mechanized divisions*") without first requiring that the underlying program contain the necessary semantic distinctions and linguistic information. We return to this example later and show how the template named in Figure 1 builds an input specification for Mumble.

MUMBLE'S PLACE IN THE GENERATION PROCESS.

A key question is what information the input specifications to Mumble represent. This amounts to asking how we take the generation process to divide into subprocesses---what decisions have already been made and are reflected in the specifications, and which ones remain. Since we have positioned the level of the specification language so as to fit the decomposition reflected in our own work and to expedite the use of Mumble-86 by other researchers, the answer can be given quite precisely. For a more complete discussion of our approach and how it contrasts with other work, see (McDonald, Meteer, & Pustejovsky, 1987). Overall we can divide the generation process into three coarse stages:

Underlying program -- Developed independently of the generator per se, this will be the expert diagnostician, cooperative database, ICAI tutor, etc. that the human users want to talk with. Some event within this underlying program will determine the goals the utterances are to achieve and initiate the generation process.

Planning -- This process determines how the goals can be achieved in a given context. This includes selecting the information to be communicated (or omitted), determining what perspectives and rhetorical organization the information should be given, and choosing a mapping for the information onto the linguistic resources that the language provides (i.e. open-class words and syntactic constructions).

Realization -- This process carries out the planner's specifications to produce an actual text. It has the responsibility for insuring that the text is grammatical, and will handle the bulk if not all of the syntactic and morphological decision making.

In these terms, Mumble-86 is a realization component.[3] As such, we expect any system that uses it to be able to supply the following kinds of information about each utterance that it wants produced, couching the information in terms of our specification language. Mumble-86 is agnostic as to whether this information was assembled by a theoretically interesting planning component or merely stipulated in predefined templates.

(a) The units from which the utterance is to be composed. The mapping for each unit to its intended linguistic resource will either have been already made or will be fully defined for later execution.

(b) The functional relationships among the units, e.g. predication, head, modifier, given, theme, etc., that direct or constrain the units' organization within the text.

(c) Lexical choice. As the primary means of delimiting what information is or is not communicated and what perspectives and connotations are presented, all open class words are choosen by the planner.

[3] We also refer to Mumble as a "linguistic component", reflecting the fact that all of the planners and underlying programs that have been used with Mumble to date have concentrated on conceptual issues and left all of the linguistic efforts to it; this designation may have to change in the coming years as the semantic and discourse level contributions of earlier components become more significant.

We see our specification language as providing a medium for the results of a planner's decisions. The syntax of the language provides a flexible, compositional notation by which a planner may view the potential linguistic form of the utterance it is constructing without having to understand the myriad details entailed by descriptions at the level of the surface structure. In the next section, we describe the syntax of the specification language. We then look at how predefined templates can be used to abstract away some of the details to make it easier for a planner to construct them.

THE INPUT SPECIFICATION LANGUAGE

Mumble's input specifications may be seen as expressions over a vocabulary of elementary terms and a syntax for their composition. In defining this language, our choice of terms and compositional operators was driven by what appears to be most useful at the linguistic level. The simplest expressions in the language, **kernel specifications**, represent the choice of a class of phrases with a lexical head and the specification of its arguments. This reflects our belief that one almost never chooses just to use a certain word, but rather to describe an action with a verb and a specific set arguments for example (see also Kegl, 1987). The result of realizing a kernel is a phrasal unit comparable to an elementary tree of a Tree Adjoining Grammar. (See Joshi, 1987, for a discussion of properties of a TAG which make them well suited to generation.) Formally, a kernel consists of a realization function and a list of arguments which are applied to it, where a realization function is typically a class of phrases distinguished by the characteristics of the syntactic contexts in which they may appear. Executing the realization function consists of choosing among the phrases and instantiating the choice.

Larger, more complex utterances are formed by composing kernels: joining them syntactically according to the relationships between them. This process is analogous to adjunction in a TAG. In Mumble, these compositional expressions are called **bundles**. They have three major parts:

(1) The **head** is either a kernel or a bundle; it is realized first, as an "initial tree" into which other specifications are attached; every bundle must have a head.

(2) **Further-specifications** have two parts, a specification (either a kernel or a bundle) and an attachment function, which constrains where the new tree may be adjoined to the surface structure already built; these correspond to the "auxiliary trees" of a TAG; a bundle may have any number of further specifications.

(3) **Accessories** contain information about language-specific syntactic details, such as tense and number. Each bundle type has a specific set of obligatory and optional accessories associated with it.

Note that bundles are not constrained as to the size of the text they produce: they may produce a single noun phrase or an entire paragraph.

Figure 2 shows a representation of the input specification for the description "*53rd Mechanized Division*" discussed at the beginning of the paper. In the next section we describe how this specification could be built from an object in the underlying program.

```
#<bundle general-np
  :head #<kernel :realization-function
                        np-common-noun
                 :arguments ("division") >
  :further-specifications
    ((:specification
          #<kernel :realization-function adjective
                   :arguments ("53rd")>
      :attachment-function restrictive-modifier)
     (:specification
          #<kernel:realization-function adjective
                  :arguments ("mechanized")>
      :attachment-function restrictive-modifier))
  :accessories (:number singular
                :gender neuter
                :person third
                :determiner-policy no-determiner)>
```

FIGURE 2

Specifications are implemented as structured objects, indicated by the "#< ... >" convention of CommonLisp; the first symbol after the "<" gives the object's type. Other symbols are either object names (e.g. "general-np"), or in a few cases print forms of whole objects (such as the accessories and their values). Strings in double quotes (e.g. "53rd") designate words.

DIRECT MAPPING: THE SIMPLE CASE

The granularity and vocabulary of the input specification language are designed to be well suited for generating natural language. In principle the semantic organization could match the structure of the specification language exactly. If this were the case, the mapping between units in the underlying application program and the specifications to the generator would be direct and one to one. However, we cannot assume that today's underlying program will have the same granularity or be able to reason in the same vocabulary. For example, while the accessories NUMBER, GENDER, and PERSON in the specification above are necessary to determine the correct pronoun, few underlying programs working with mechanized divisions would bother to represent their gender. Rather than force a planner to deal in these terms, we provide a framework for building specifications piecemeal by applying **templates** that can be specialized to the application. Templates are abstractions of specifications, which stipulate some of the terms in the specification and parameterize others. An object in the underlying program may be mapped to a template through a default specification, as illustrated in Figure 1 and repeated below along with the template ARMED -FORCES-UNIT-NAME:

```
(define-default-specification
                '53rd-mechanized-division
  :template-name armed-forces-unit-name
  :arguments ("53rd"  "Mechanized"  "Division") )

(define-specification-template
                        armed-forces-unit-name
                        (number type size)
  (let ((K (make-a-kernel 'np-common-noun size))
       (B (make-a-bundle 'general-np)))
    (set-bundle-head B K)
    (neuter-&-third-person B)
    (singular B)
    (no-determiner B)
    (add-specializing-description
       (property-realized-as-an-adjective number)
       B)
    (add-specializing-description
       (property-realized-as-an-adjective type)
       B)
    B))
```

FIGURE 3

As a formal entity, this template is essentially a procedure for assembling the data structures that make up a specification. It is a Lisp program and draws on a set of predefined functions (e.g. set-bundle-head, no-determiner) to simplify the statement of the necessary actions. Every template is required to provide all of the elements that make up a properly formed realization specification. In this case a bundle for a noun phrase is being assembled, and so there must be a kernel built for the head of the bundle and values given for all the accessories that bundles of that type require. Since the phrases specified by this particular template are compositions linguistically, i.e. they involve the adjunction of two modifiers to the inital np-common-noun, the template includes operations ("add-specializing-description") that add the sources of the modifiers using the proper attachment function.

These same techniques may be used to generate longer texts. The following example differs from the last one in three ways:

(1) The templates are building larger structures: discourse units which produce multiple sentences and clause bundles which produce complex sentences.

(2) Default mappings are defined between classes of objects and templates rather than having to define a mapping for each instance of the class.

(3) Templates can be called explicitly from other templates with a dynamically chosen set of arguments.

The example is from one of the generation tasks in the ALBM domain: to produce a "mission restatement" paragraph describing the essential tasks in some operation. These tasks are presented to the generator as a simple list of TASK-OBJECTS, expressing the who, what, when, where, and why of the task, along with a dependency graph representing the relations between them. Figure 4 shows an example of a task object and a portion of a mission restatment paragraph produced by our current prototype of the text planner.

```
#<unit TO1.ATTACK[4]
  parent: #<unit TASK.OBJECT>
  slots:
    unit: #<unit 10TH-CORP>
    action: #<unit ATTACK>
    objective: #<unit NORTHEAST>
    intent: #<unit SECURE.OBJECTIVES> >
```

"10th (U.S.) Corps attacks to the northeast to secure objectives. 10th (U.S.) Corps exploits east of Thuringer Wald."

FIGURE 4

4 For brevity and clarity we use a textbook frame style rather than showing the actual KEE underlying representation; we also show only the slots which directly impact this discussion.

Our prototype text planner takes advantage of the uniformity of the objects in the underlying program that motivates the text and the uniformity in the form of the paragraphs to be produced. These uniformities allow us to use predefined templates for these paragraphs in much the same way as McKeown used schemas to produce the overall organization of definitions of data base attributes (McKeown, 1985). Note that there are two very important assumptions inherent in this approach: First, the information needed is explicitly represented in data structures in the underlying program. Second, those data structures are stable, that is, in the lifetime of the project, the structures will not change, or if they do, then the specific templates that access them must change as well.

```
(define-default-specification
                        (k:unit 'K::task.object)
  :template-name express-task
  :arguments ())


(define-specification-template express-task5  ()
  (let* ((r-fn (instantiate-mapping
                (k:unit
                 (k:get.value self 'k::action))))
         (agent (instantiate-mapping
                 (k:unit
                   (k:get.value self 'k::unit))))
         (k (make-a-kernel r-fn agent))
         (loc (when
                (k:get.value self 'K::objective)
                (make-a-further-specification
                  'location-modifier
                  (instantiate-mapping
                    (k:unit
                      (k:get.value self
                                   'K::objective)))
              )))
         (intent (when
                   (k:get.value self 'K::intent)
                   (make-a-further-specification
                     'rationale-modifier
                     (k:unit
                       (k:get.value self
                                    'K::intent))
                 ))))
    (funcall-template 'current-event-with-modifiers
                      k loc intent)))
```

FIGURE 5

[5] *Self* is bound to the instance being mapped at the time the mapping occurs; in this case, it is bound to the unit to1.attack.

The top level function for generating the mission paragraph builds a discourse unit bundle with the first task-object as the head of the bundle and the rest as an ordered list of further-specifications. Since the relations between the task objects in this example is simply sequential-temporal, the default attachment function "new sentence" is used, resulting a sequence of separate sentences, one for each task.

Figure 5 shows the default specifications for the class task-object and the template it references.

This template is a specialist which picks out the information from the task object to be included in the mission paragraph. Note that the modularity of the task objects is different from that of the actual sentences which express them. The action and unit combine to form the matrix of the sentence and other slots function as adjuncts, such as the location and intent. The template shown in Figure 6 combines these elements into a clause bundle and sets the accessories to unmarked (not a question or command) and simple present tense. These features are stipulated as part of the style of these paragraphs rather than stemming from anything in the underlying representation.

```
(define-template current-event-with-modifiers
                 (event &rest modifiers)
 (let ((b (make-a-bundle 'general-clause))
   (set-bundle-head b event)
   (present-tense b)
   (unmarked b)
   (dolist (m modifers)
     (add-already-built-further-specification m b)
  b))
```

FIGURE 6

In the examples described above, our use of templates is a shorthand for building realization specifications. As such it is appropriate for the very simple text planning that typifies today's generation applications: Already formed objects and expressions in the underlying application program can be associated directly with semi-custom templates with the English words introduced as arguments. In more complex text planning where, for example, the same objects are presented from different perspectives depending on the communicative situation, there is unlikely to already be any expression with the right properties, and it will be the planner's task to construct one. Here too, our facility for mapping objects to specifications will be very useful.

COMPOSING SPECIFICATIONS

In this section we look ahead to the development of general planners with the ability to dynamically select and orchestrate information from the underlying program to fit the occasion. One of a planner's prime abilities will be to appreciate the functions and consequences of alternative forms and combinations by which the same body of information can be communicated. Our specification language permits such alternatives to be simply stated. We can see this in an illustration taken from our ongoing work with the KRS system in use at the Rome Air Development Center. KRS ("Chris") is a rule based system for mission planning. Its internal representation is based on instantiating relations represented as lists of symbols: for example the three relations shown below in Figure 7 ("facts" in the lefthand side of one of KRS's production rules), along with their English realizations as given by the direct replacement generator presently included with KRS.

```
(target OCA1002 BE50318)
(POWA BE50318 BE50318-Search-Radar)
(IS-A BE50318-Search-Radar  Electronics)
```

The target of OCA1002 is BE50318. Part of BE50318 is BE50318-Search-Radar. BE50318-Search-Radar radiates.

FIGURE 7

While perhaps good enough to serve its purpose (i.e. as part of the KRS rule-editor), this text is unnatural--no person would ever say it. Stylistically it is chunky and awkward, but more importantly, it actually mis-communicates the relative value of the three facts by giving them equal weight in the utterance.

A sophisticated text planner would want to convey not just propositional information but also to indicate its rhetorical significance, e.g. what is important, what is unusual. In the present case, the fact about the assignment of the target was specified by the user and is thus a given. The fact that the target has a search radar may or may not already be known. The fact that this particular radar is known to be active is the most significant, since it is this fact that has an impact on the planning of the mission (i.e. there now have to be radar-suppression aircraft included).

Depending on whether or not the existence of the search radar is known, a much improved rendering of the three facts could be one of these two:

"The target has an active search radar"
"The target's search radar is active"

Given that we will have already established mappings to suitable templates for each of the three facts independently, the specification of a single sentence expressing all three becomes a matter of combining them into a single specification, varying their positions as bundle heads or further specifications and specifying the appropriate attachment functions. This ability to combine parts without affecting their internal structure is one of the most powerful aspects of our specification language.

The specification for *"The target has an active search radar"* (Figure 8) would be built by using the second fact, "part of", as the backbone of the bundle, supplying the head and thereby the main verb *has.* The first fact--`(target ... BE50318)`--is then folded in as the way of describing BE50318 (interpreting the fact as ascribing the functional role of "target" to its second argument, the "battle element"), and the third fact--`(isa ... electronics)`--becomes a modifier in the description of the search radar.

Alternatively, to specify *"The target's search radar is active"* (Figure 9), one would position the third fact as the head of the bundle and use the first two as the characterization of the search radar. As indicated on the two figures, these specifications are assembled from exactly the same three partial specifications, but combined in different orders with different attachment functions.

Note that the pretty-printing of these specifications is a little simpler than the earlier ones so as to conserve space, and that it includes another field--"underlying-object"--to make the origins of the different parts of the specification clearer.

One other point that may be unexpected is the fact that the Figures include <u>two</u> instances of the specification for the search radar, one as the second argument to *have* as we would expect, and a second embedded within the first as part of the "clause" specification for `(ISA ... electronics)`. Of course, if this second instance were missing--say as the result of some planning-level abbreviation in recognition that only the adjective within that specification was going to actually appear in the final text--then the specifications in the two figures would not just be simple rearrangements of the same parts (a generalization we consider valuable); instead we have the selection of the adjective done as part of realization as one of the normal choices for simple predications, under control of the position where the specification is attached, i.e. as a modifier to an NP.

```
#<bundle  general-clause
 :underlying-object (POWA ... search-radar)
 :head
  #<kernel HAVE-as-possession
     ( #<bundle general-np
         :underlying-object BE50318
         :head #<kernel NP-common-noun
                        ("target")>
         :accessories (:number singular
                       :gender neuter
                       :person third
                       :determiner-policy
                              always-definite)>
       #<bundle general-np
         :underlying-object BE50318-search-radar
         :head #<kernel NP-common-noun
                        ("radar")>
         :accessories (:number singular
                       :gender neuter
                       :person third
                       :determiner-policy
                              always-definite)
         :further-specifications
          ((:specification
             #<kernel common-noun
                      ("search")>
            :attachment-function  classifier)
           (:specification
             #<bundle general-clause
               :underlying-object (ISA...electronics)
               :head
                 #<kernel predication_to-be
                    (#<bundle general-np
                       :underlying-object
                                  BE50318-search-radar
                       :head #<kernel NP-common-noun
                                      ("radar")>
                       :accessories (:number singular
                                     :gender neuter
                                     :person third
                                     :determiner-policy
                                       always-definite)
                       :further-specifications
                        ((:specification
                           #<kernel common-noun
                                    ("search")>
                          :attachment-function
                                      classifier))>
                     #<kernel ADJP-adjective
                              ("active")> ) >>
           :attachment-function restrictive-modifier
           ))> )>
 :accessories
    (:unmarked
     :tense-modal present)>
```

"The target has an active search radar."

FIGURE 8

```
#<bundle general-clause
 :underlying-object (ISA...electronics)
 :head
  #<kernel predication_to-be
    ( #<bundle general-np
        :underlying-object BE50318-search-radar
        :head
         #<kernel NP-common-noun
                  ("radar")>
        :accessories (:number singular
                      :gender neuter
                      :person third
                      :determiner-policy
                             always-definite)
        :further-specifications
         ((:specification
            #<kernel common-noun
                     ("search")>
           :attachment-function  classifier))
          (:specification
            #<kernel HAVE-as-possession
               ( #<bundle general-np
                   :underlying-object BE50318
                   :head #<kernel NP-common-noun
                                  ("target")>
                   :accessories (:number singular
                                 :gender neuter
                                 :person third
                                 :determiner-policy
                                   always-definite)>
                 #<bundle general-np
                   :underlying-object
                               BE50318-search-radar
                   :head #<kernel NP-common-noun
                                  ("radar")>
                   :accessories (:number singular
                                 :gender neuter
                                 :person third
                                 :determiner-policy
                                   always-definite)
                   :further-specifications
                    ((:specification
                       #<kernel common-noun
                                ("search")>
                      :attachment-function
                                   classifier))
           :attachment-function  possessive ))>

      #<kernel ADJP-adjective
               ("active")> )>
 :accessories
    (:unmarked
     :tense-modal present)>
```

"The target's search radar is active."

FIGURE 9

CONCLUSION

The specification language has been completely implemented and used in-house since the fall of 1986. The templates and the specifics of how objects and expressions in underlying applications and planners are to be linked to Mumble-86 have evolved over that time and may continue to evolve somewhat as we get more experience with other applications. Mumble-86 itself is currently being used both for applications and as a research tool at a variety of sites including the University of Massachusetts, BBN Labs, RADC, and University of Pennsylvania.

As part of an excercise in learning how to use Mumble-86, two researchers from RADC, Sharon Walter and Doug White, recently extended the program to generate in KRS's domain. It took them only two days to learn the specification language, build input specifications, and make the necessary grammatical and lexical extensions to generate several sentences in their domain, including those shown in Figure 7. Neither had used Mumble-86 before.

In conclusion we would like to emphasize two main points. The first is the importance of modularity in design and portability of the modules so that research can concentrate on new and hard problems without having to waste effort reinventing the wheel. Mumble-86 has been developed to be just such a portable module. It has the responsibility for all syntactic decisions without making presumptions about the semantic model of the application program that uses it.

Our second point is that a designer should not compromise the integrity of a well developed module to accomodate one which is less well developed when the two are brought together in the same system. This is the purpose of the input specification language we have introduced in this paper. In developing this language, we have clarified what decisions have to be made outside Mumble-86 and which decisions are its responsibility, thus circumscribing its sphere of influence and making it more useful as a domain independent linguistic component and as a tool for research in text planning and discourse structure.

REFERENCES

Joshi, Aravind K. (1987a) "The Relevance of Tree Adjoining Grammar to Generation", in Kempen (ed.), p. 233-252.

Joshi, Aravind K. (1987b) "Word-Order Variation in Natural Language Generation", AAAI-87 Proceedings, p. 550-555.

Kegl, Judy, (1987) "The Boundary Between Word Knowledge and World Knowledge", Proceedings of Theoretical Issues in Natural Language Processing", Memoranda in Computer and Cognitive Science, New Mexico State University, p. 26-31.

Kempen (ed.) (1987) *Natural Language Generation* , Martinus Nijoff Publishers, Dordrecht, The Netherlands.

McDonald, David D., Marie W. Meteer, & James D. Pustejovsky (1987) "Factors Contributing to Efficiency in Natural Language Generation", in Kempen (ed.), p. 159-182.

McKeown, Kathleen R. (1985) "Discourse Strategies for Generating Natural Language Text", *Artificial Intelligence* 27, p. 1-42.

Meteer, Marie W., David D. McDonald, Scott Anderson, David Forster, Linda Gay, Alison Heuttner, & Penelope Sibun, "Mumble-86: Design and Implementation", UMass Technical Report 87-87, 173 pages.