A Natural Language Generation Component for Dialog Systems

Susan W. McRoy[1] **Songsak Channarukul**[1] **Syed S. Ali**[2]
mcroy@uwm.edu, songsak@uwm.edu, syali@uwm.edu
[1]Electrical Engineering and Computer Science
[2]Mathematical Sciences
University of Wisconsin-Milwaukee
3200 N. Cramer Street, Milwaukee, WI 53211

Abstract

We present a natural language generation component, called YAG, that is suitable for dialog systems. Dialog systems collaborate with their users in real-time during the course of an interaction. A natural language interaction (dialog) makes mixed-initiative systems more realistic since one of communication media that people are comfortable with is a natural language. YAG allows a mixed-initiative system to parameterize templates differently to suit each utterance it makes depending on its role or communicative intention in dialogs. YAG provides real-time generation of text from the knowledge represented in an application. Our approach to real-time generation in dialog systems combines a template-based approach for the representation of text structure with knowledge-based methods for representing semantic content. In this approach, a dialog system specifies a template and content from its knowledge base to realize the text. The content can be specified in one of two ways: (1) as a sequence of propositions (2) as a set of feature value pairs. To accomplish the realization, YAG instantiates the specified template with the given content to produce the text.

Introduction

Many dialog systems have tried to allow natural, flexible communication between people and computers by supporting natural language as a medium of communication. To support dialog that adapts to the user in real-time, a natural language generation component must be fast, robust, extensible, and maintainable. Although a number of researchers have worked on building natural language generation systems, none of these systems meet the needs of dialog systems. Some approaches provide a broad coverage of English grammar, but are better suited to off-line generation, because speed is not a concern. Other approaches address speed requirements, but do so by focusing on a specific domain. Grammar and other linguistic knowledge needed by the generation system is not declarative; this makes it difficult to modify, extend, or maintain them.

YAG (**Y**et **A**nother **G**enerator) provides an alternative way of generating natural language text by combining a template-based approach for the representation of text structure with knowledge-based methods for representing semantic content. YAG's inputs are concepts or propositions along with optional annotations to specify syntactic constraints. A *template* is a pre-defined form that is filled by the information specified by either the user or the application at run-time. The template-based approach avoids unification of a grammar with an input structure, which can be a time-consuming process (in part because many alternatives must be considered). YAG avoids this problem by requiring the application that uses it to specify appropriate templates for the given content. Further, templates are accessed using a hash table, which facilitates fast access. Additionally, input to YAG need only be partially specified; YAG will uses default values to fill in any missing feature values. YAG templates are declarative, comprehensible, easy to modify, extend, and maintain. Thus, YAG is a real-time generation system that supports natural and efficient interaction in dialog systems. Templates in YAG are flexible so that a mixed-initiative system can parameterize them differently to suit each utterance it makes depending on its role in dialogs. (Conditions in template rules enable YAG to generate different texts based on the values given to a template.)

YAG can realize a text from two different sources of input: a sequence of propositions in a logical language or a feature structure along with the name of a template. These two realization processes will be explained in the following sections.

A Template-based Approach

YAG combines a template-based approach for the representation of text with knowledge-based methods for representing content. Each form in the template (a list) is a rule that expresses how to realize a surface constituent. Figure 1 illustrates an example template from YAG. This template would be used to generate a sentence for expressing propositions of the form `has-property(agent, pname, pval)`, such as

```
has-property(John, age, 20).
```

```
((C ((O (E agent (subjective))
        (equal pname nil))
     (O ((E agent (possessive))
         (F pname))
        (not (equal pname nil)))))
 (V "be" ((subject agent)
          (tense present)))
 (C ((O (F property)
        (not (equal property nil)))
     (O (F pval)
        (not (equal pval nil)))))
 (S ".")
)
```

Figure 1: An Example Template

Templates are realized as strings by replacing each variable in a rule with an instantiated value and then evaluating the rule. In the template in Figure 1, if **agent** = "John", **pname** = "age", and **pval** = "20", the surface text will be "John's age is 20.". The template has four parts, for generating the subject, verb, property, and punctuation, in order. The first rule, which is a condition rule, has two alternatives. (Such alternatives are called *options*; in each option, the condition is given last.) In the first rule, the second option is chosen because **pname** has been specified. Within this option, the **agent** is generated as a possessive, "John's", followed by the value of the feature **pname** (which is the string "age"). Next, the verb rule is executed. The verb "be" together with its features, **subject** = agent and **tense** = present generates the verb "is". The third rule is another condition rule. In this rule, the first option fails, because no **property** is specified. The second option applies because **pval** has been specified (**pval** = "20"). Thus, this third rule generates the value of the feature **pval** (i.e. "20"). The final rule always returns the period punctuation. Finally, all outputs are concatenated in order and returned as a string "John's age is 20.".

A complete description of how to define templates is given in the YAG User's Manual (Channarukul 1999).

Natural Language Generation from Knowledge Representation

YAG provides facilities for generation from any knowledge representation language. This is accomplished by the use of a knowledge representation specific component which must be defined for that particular knowledge representation language. (In the architecture of YAG, shown in Figure 5, this component corresponds to surface-2.) This component maps the specific knowledge representation into a feature structure that can be processed by the domain and knowledge representation independent component of YAG (shown as surface-1 in Figure 5).

The knowledge representation that is currently being used is SNePS, which uses propositional semantic networks (Shapiro & Group 1998). A *propositional semantic network* is a framework for representing the concepts of a cognitive agent who is capable of using language. The information is represented as a labeled, directed graph that represents relations (defined by its arcs) to entities (defined by its nodes).

Figure 2 shows an example of a propositional semantic network. In the network, the node M2 represents the proposition that the discourse entity B2 is a member of class *"dog"*. The node M5 represents the proposition that the name of the discourse entity B2 is *"Pluto"*. We can read the whole proposition as *"Pluto is a member of class dog."* or simply *"Pluto is a dog."*.

```
((M2 (CLASS "dog")
     (MEMBER B2))
 (M5 (OBJECT B2)
     (PROPERNAME "Pluto")))
```

Figure 2: An Example of a Semantic Network.

Case frames, which are conventionally agreed upon sets of arcs emanating from a node, are used to represent propositions in a semantic network. For example, to express that A *isa* B we use the MEMBER-CLASS case frame which is a node with a MEMBER arc and a CLASS arc. YAG realizes text from a proposition. More complex text can be realized by providing multiple propositions. Example 1 illustrates an input to YAG.

Example 1 *Pluto is a dog.*

```
'(((M2 (CLASS "dog")
       (MEMBER B2))
   (M5 (OBJECT B2)
       (PROPERNAME "Pluto")))
  ((form decl)
   (attitude be) )
 )
```

In Example 1, the input includes two propositions and a list of control features. The first proposition is the primary proposition to be realized (that is the surface sentence will be about something being a dog rather than something being named *"Pluto"*). M2 says that the discourse entity B2 is a member of class *"dog"*.

The second proposition (M5) says that *"Pluto"* is the name of the entity B2. Together, these propositions say that the entity B2, whose name is *"Pluto"*, is a member of class *"dog"*. YAG will map the MEMBER-CLASS proposition to the template shown in Figure 3. The control features, **form** = decl and **attitude** = be, are also used in selecting the exact template to be used. (For example, if the form had been interrogative, a different template would have been used.)

Prior to realization, the knowledge engineer must provide a mapping from each case frame to the name of the corresponding template in a *case frame mapping table*. (This is the primary task in constructing a new surface-2 component for the knowledge representation.) A mapping table is an index table in which a case frame and selected control features serve as an index to a specific template. In this example, YAG selects the member-class template to realize the information extracted from the given semantic network.

```
'((E member)
  (V "be" ((subject member)))
  (E class))
```

Figure 3: A member-class Template.

Example 2 shows how syntactic constraints can be added to override defaults made by YAG. Here, to produce a definite noun phrase (*"the book"*), we override the definiteness default, which is NO, for the noun *"book"* which is in the third proposition (M11) in the example. (This is done by adding the control feature: (definite YES B6)).

Example 2 *"George reads the book."*

```
'(((M2 (ACT (M1 (ACTION "read")
                (DOBJECT B6)))
       (AGENT B4))
   (M5 (OBJECT B4)
       (PROPERNAME "George"))
   (M11 (CLASS "book")
        (MEMBER B6))
  ((form decl)
   (attitude action)
   (definite YES B6)))
 )
```

Default feature values can be overridden for any entity by adding to the content specification a list that contains the name of feature, its new value, and the node corresponding to the entity (in Example 2, this is B6). Pronominalization can also be done this way, as shown in Example 3.

Example 3 *"He understands it."*

```
'(((M2 (AGENT B4)
       (ACT (M1 (ACTION "understand")
                (DOBJECT B6))))
   (M5 (OBJECT B4)
       (PROPERNAME "George"))
   (M11 (CLASS "book")
        (MEMBER B6))
  ((form decl)
   (attitude action)
   (pronominal YES (B6 B4))
   (gender MASCULINE B4)))
 )
```

In Example 3, AGENT-ACT is the primary proposition in the semantic network. In this example, the proposition says that there is the agent (B4) who is doing the action *"understand"* on the object (B6). This proposition along with the selected control features (**form** = decl and **attitude** = action), allows YAG to select the clause template.

To override the **gender** default (neutral) of B4 and generate *"He"* instead of *"It"*, Example 3 specifies B4's **gender** as MASCULINE. To override the default expression type (full noun phrase) for both B4 and B6, Example 3 specifies (pronominal YES (B6 B4)) which forces pronominalization.

The next section describes the realization from a feature structure representation.

Natural Language Generation from Feature Structures

As previously mentioned, YAG is independent of the underlying knowledge representation (here, SNePS) because it also accepts feature structures as input. A feature structure is composed of one or more features and their values. Each pair in a feature structure is a list containing the feature's name as the first item and its value as the second. The value can be a string, a symbol, or another feature structure. Example 4 shows a complete feature structure that would be used to realize the text *"John walks."*.

Within a feature structure, the name of the template that YAG should use is given by the template feature. YAG then selects the template from one of its template libraries and realizes the template as discussed earlier in Section . Template rules are realized in sequence. When a template rule contains a variable, the value is obtained from the feature structure.

Example 4 *"John walks."*

```
'((template clause)
  (agent "John")
  (process "walk"))
```

In Example 4, YAG retrieves the `clause` template[1] which is shown in Figure 4.

```
'(rules ((E agent)
         (V (process) ((subject agent)))
         (S "."))
 )
```

Figure 4: A simplified template rule of `clause` template.

The rule `(E agent)` is realized first, yielding *"John"* as its output. Then, the rule `(V (process) ((subject agent)))` is processed. In this template rule, the value bound to `agent` is a subject of the verb stored in the feature **process**. In this example, the value of `agent` feature is a string *"John"*. The verb is *"walk"*. To correctly process the verb, the default values of `number`, `person` and `tense` are applied. (The default values of these features in the verb template result in the inflected form, *"walks"*.) The partial result is *"John walks"*. Finally, the partial result is concatenated with another string, a period ".", yielding the final output *"John walks."*.

The next example shows the feature structure representation that would be used to generate the string *"He understands it."*.

Example 5 *"He understands it."*

```
((template clause)
 (agent ((template noun-phrase)
         (case proper)
         (head "George")
         (gender masculine)
         (pronominal yes)))
 (process "understand")
 (object ((template noun-phrase)
          (case common)
          (head "book")
          (pronominal yes))) )
```

In this example, the feature structure illustrates that the value of a feature may be another feature structure, as is the case for the features **agent** and **object**. Allowing a feature structure to be embedded in another feature structure increases the flexibility of YAG.

In this example, the embedded feature structure

```
((template noun-phrase)
 (case proper)
 (head "George")
 (gender masculine)
 (pronominal yes))
```

would be realized first, returning the word *"He"*, because of the **pronominal** and **gender** constraints. Then, the `agent` feature would be bound to *"He"* in the `clause` template. The value of the `process` feature is given as a string. (The template specifies that the default **number**, **person**, and **tense** are `singular`, `third-person`, and `present` respectively.) Thus, the **process** feature is realized as *"understands"*. The value of the **object** feature is then realized as the word *"it"*. Finally, the `clause` template is realized as the string *"He understands it."*.

System Architecture

Our implementation of YAG has a layered architecture as shown in Figure 5. It allows an application to realize texts from two kinds of input, a knowledge representation or a feature structure. These realization processes have been described in the previous sections. This section will briefly describe the architecture of YAG as a whole system.

The **outer layer** (`surface-2`) of YAG realizes a given semantic network. The input contains two parts: a semantic network that represents content, and a set of control features that provide supporting information and optional syntactic constraints. Some of these control features are used by YAG to select the appropriate template, the remainder are used to select options within a template. The output of this layer is a feature structure that is passed to the inner layer for further realization into a string of words.

The **inner layer** (`surface-1`) accepts a feature structure as input. This feature structure specifies the template to be used along with other features and their values. In addition, this layer will use defaults to specify any missing values in a feature structure input. The inner layer processes its input by applying templates in a depth-first manner.

The **Lexicon** contains word level information. Templates can access the lexicon directly with a template rule. YAG includes lexical functions to inflect a given verb according to verb features (*e.g.,* tense, person, aspect, etc.), and to generate the singular or plural forms of nouns. Additional lexicon functions can be coded if required.

[1] This template has been simplified to facilitate explanation.

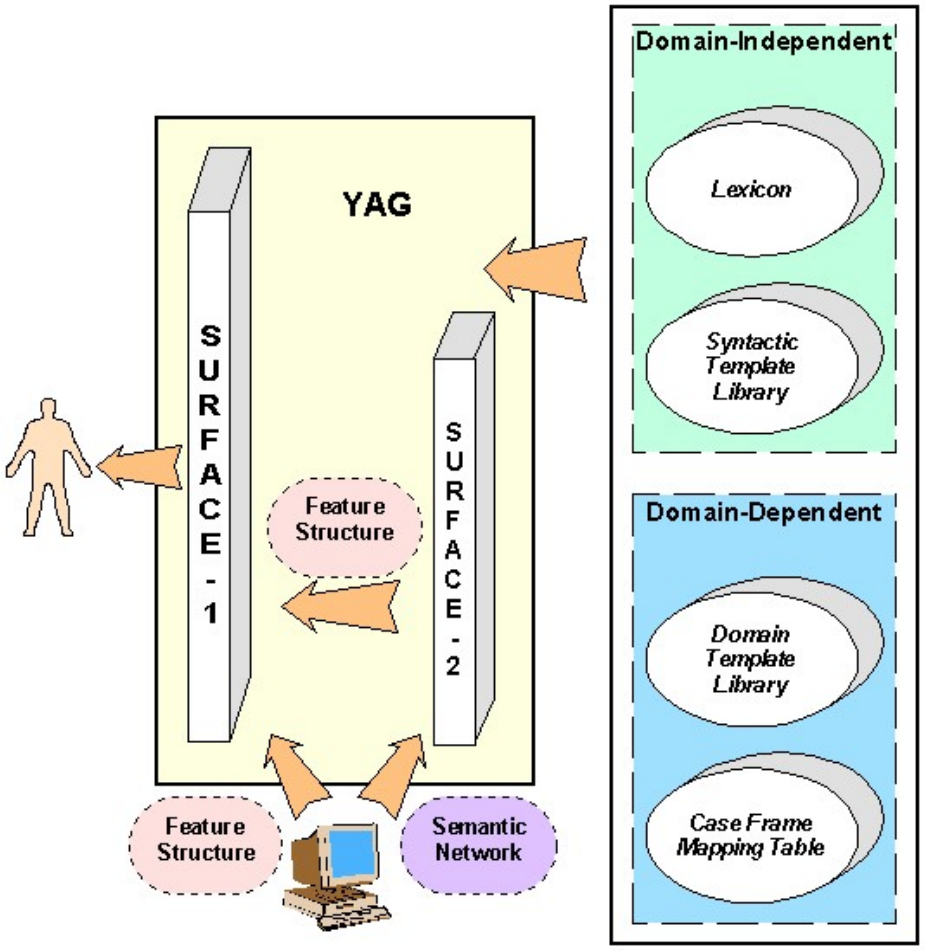

Figure 5: Architecture of YAG

The **Syntactic Template Library** contains templates that are used as a grammar of English. Other templates can embed these templates to form more complex structures. They may also be combined with templates from the domain template library. (The current implementation of YAG includes a number of predefined syntactic templates.)

The **Domain Template Library** contains templates that are specific to a particular application. Developers can author their own templates when necessary. These templates can serve applications that have special[2] information presentation needs.

The **Case Frame Mapping Table** is used to link the semantic case frames with their associated templates. The outer layer will access the mapping table with the selected control features to pick to the appropriate template when realizing text from the knowledge representation.

[2]By special, we mean that a sublanguage is appropriate, for example *e.g.,* scientific information, weather reports, etc.

Related Work

Although a number of researchers have worked on building natural language generation systems, none of this work meets the needs of dialog systems. TEXT (McKeown 1982; 1985) uses a template as a plan. Templates are realized by a separate module that uses a functional unification grammar. Speed is an issue in unification-based generation approaches; this is a problem in real-time dialog. Moreover, templates in TEXT primarily address the generation of a complete discourse (multiple sentences or a paragraph). However, in dialog, agents communicate information to each other incrementally. YAG meets the needs of incremental dialog because it can generate text at any length, including a single word or a sentence, or multiple sentences.

FUF/SURGE (Elhadad 1992; 1993) is a feature-based system that uses a functional unification grammar to realize text. FUF is a surface realization component. SURGE is a declarative grammar written in FUF's formalism. SURGE's coverage of grammar is

very broad, thus the quality of generated text is high. However, the unification process employed by FUF dramatically decreases its speed when the grammar becomes large. YAG provides an alternative that is appropriate when generation speed is crucial and fine-grained control of text is not necessary.

Penman/Nigel (Mann 1983) covers broader tasks of generation (content determination, discourse planning, surface realization, and text revision). Penman is a feature-based generator with a systemic grammar (called Nigel). Although it does not use grammar unification, it is similar to FUF/SURGE in that it must traverse a grammar to generate texts. The speed of generation thus decreases as the size and complexity of the grammar increases.

MUMBLE (Meteer *et al.* 1987) is a phrase-based, surface realization component. Its input is a feature structure that specifies a phrase to be used. A realization component realizes the final surface string by instantiating a phrase structure with conceptual information from the input. YAG's syntactic templates are similar to the phrase-based grammar of MUMBLE. An additional advantage of YAG is that YAG's templates can be defined at different levels of abstraction, not only at the syntactic, but also at the semantic level.

RealPro (Lavoie & Rambow 1997) has been designed with speed in mind. It is a surface realization component that has a declarative grammar and lexicon. Its drawbacks are similar to those generation systems that limit the generation process to syntactic level information, thus generating natural language from a knowledge representation may be difficult.

Conclusion

We have presented the natural language generation component, called **YAG** (**Y**et **A**nother **G**enerator), that has been designed to meet the needs of real-time dialog systems. YAG has been integrated into B2, a general-purpose tutoring system that allows students to practice their decision-making skills in a number of domain (McRoy 1998; McRoy, Haller, & Ali 1997; 1998). B2 supports mixed-initiative interaction using a combination of typed English utterances and point-and-click-based communication using a mouse.

YAG combines a fast template-based approach for the representation of text structures with knowledge-based methods for representing content. Its inputs are concepts or propositions along with optional annotations to specify syntactic constraints, thus allowing the generation of natural language from knowledge representation. YAG improves practical system abilities in natural language generation by supporting fast and incremental interaction for real-time dialog systems.

References

Channarukul, S. 1999. *The YAG's User Manual.* Department of Computer Science, University of Wisconsin-Milwaukee.

Elhadad, M. 1992. *Using argumentation to control lexical choice: A functional unification-based approach.* Ph.D. Dissertation, Computer Science Department, Columbia University.

Elhadad, M. 1993. FUF: The universal unifier - user manual, version 5.2. Technical Report CUCS-038-91, Columbia University.

Grosz, B. J.; Sparck-Jones, K.; and Webber, B. L. 1986. *Readings in Natural Language Processing.* Los Altos, CA: Morgan Kaufmann Publishers.

Lavoie, B., and Rambow, O. 1997. A fast and portable realizer for text generation systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, 265–268.

Mann, W. C. 1983. An overview of the Penman text generation system. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, 261–265. Also appears as USC/Information Sciences Institute Tech Report RR-83-114.

McKeown, K. R. 1982. The TEXT system for natural language generation : An overview. In *Proceedings of the 20th Annual Meeting of the ACL*, 113–120.

McKeown, K. R. 1985. Discourse strategies for generating natural-language text. *Artificial Intelligence* 27(1):1–42. Also appears in (Grosz, Sparck-Jones, & Webber 1986), pages 479-499.

McRoy, S.; Haller, S.; and Ali, S. S. 1997. Uniform Knowledge Representation for NLP in the B2 System. *Natural Language Engineering* 3(2):123–145.

McRoy, S. W.; Haller, S. M.; and Ali, S. S. 1998. Mixed-Depth Representations for Dialog Processing. In *Proceedings of Cognitive Science '98*, 687–692. Lawrence Erlbaum Associates.

McRoy, S. 1998. Achieving Robust Human-Computer Communication. *International Journal of Human-Computer Studies* 48(5):681–704.

Meteer, M. W.; McDonald, D. D.; Anderson, S. D.; Forster, D.; Gay, L. S.; Huettner, A. K.; and Sibun, P. 1987. Mumble-86: Design and implementation. Technical Report COINS 87-87, Computer and Information Science, University of Massachusetts at Amherst.

Shapiro, S. C., and Group, T. S. I. 1998. *SNePS 2.4 User's Manual.* Department of Computer Science, SUNY at Buffalo.