YAG : A Template-Based Generator for Real-Time Systems*

Susan W. McRoy **Songsak Channarukul** **Syed S. Ali**
{*mcroy, songsak, syali*}*@cs.uwm.edu*

Natural Language and Knowledge Representation Research Group
http://tigger.cs.uwm.edu/~nlkrrg
Electrical Engineering and Computer Science Department
University of Wisconsin-Milwaukee

1 Introduction

YAG (**Y**et **A**nother **G**enerator) is a real-time, general-purpose, template-based generation system that will enable interactive applications to adapt natural language output to the interactive context without requiring developers to write all possible output strings ahead of time or to embed extensive knowledge of the grammar of the target language in the application. Currently, designers of interactive systems who might wish to include dynamically generated text face a number of barriers; for example designers must decide (1) How hard will it be to link the application to the generator? (2) Will the generator be fast enough? (3) How much linguistic information will the application need to provide in order to get reasonable quality output? (5) How much effort will be required to write a generation grammar that covers all the potential outputs of the application? The design and implementation of YAG is intended to address each of these concerns. In particular, YAG offers the following benefits to applications and application designers:

Support for Underspecified Inputs YAG supports knowledge-based systems by accepting two types of inputs: applications can either provide a feature structure (a set of feature-value pairs) or provide a syntactically underspecified semantic structure that YAG will map onto a feature-based representation for realization. YAG also provides an opportunity for an application to add syntactic constraints, such as whether to express a proposition as a question rather than a statement, as a noun-phrase rather than as a sentence, or as a pronoun rather than a full noun phrase.

Speed YAG has been designed to work in real-time. The YAG template processing engine does not use search to realize text, thus the speed of generation depends on the complexity of the template that the application selects, not on the size of the grammar. Short, simple utterances are always realized faster than longer ones. (In many other approaches, speed is a function of the grammar size, because it is searched during realization (Elhadad, 1992; Elhadad, 1993; Mann, 1983; McKeown, 1982; McKeown, 1985).)

Robustness In YAG, the realization of a template cannot fail. Even if there are inconsistencies in its input (such as subject-verb disagreement), the generator will produce an understandable (if not grammatical) output. Applications that need to enforce grammaticality can use the YAG preprocessor to detect missing or conflicting features and to supply acceptable values. The preprocessor makes use of a declarative specification of slot constraints, based on an attribute grammar (Channarukul et al., 2000). This specification is modifiable and extensible by the application designer.

Expressiveness YAG offers an expressive language for specifying a generation grammar. This language can express units as small as a word or as large as a document equally well. Unlike the typical template-based approach, the values used to instantiate slots are not limited to simple strings, but can include a variety of structures, including conditional expressions or references to other templates. (This paper will include a more detailed discussion in the next section.) Any declarative grammar, such as one based on feature structures, would be expressible in YAG.

Coverage The coverage of YAG depends on the number of templates that have been defined in its specification language. In theory, any sentence may be realized given an appropriate template. In practice, an application builder must be concerned with whether it is possible to reuse existing templates or whether it is necessary to create new ones. YAG simplifies the task of specifying a generation grammar in several ways:

*This work has been supported by the National Science Foundation, under grants IRI-9701617 and IRI-9523666, and by Intel Corporation.

- It provides an expressive, declarative language for specifying templates. This language supports template re-use by allowing template slots to be filled by other templates.
- It includes a general-purpose, template-based grammar for a core fragment of English. These templates include default values for many of the slots, so an application may omit a feature if it has no information about it. Currently, the YAG distribution includes about 30 domain-independent syntactic templates, along with some semantic templates.
- It offers a tool for helping people edit templates and see what text would be realized from a template, given a set of values for its slots.

YAG itself comes in two versions, one in CLISP, one in JAVA, both of which run on a wide variety of platforms, including Unix and Windows 95/98.

In the remainder of this paper, we will describe YAG's template specification language, and some examples that illustrate the use of YAG from an application. More details can be found in (Channarukul, 1999).

2 YAG's Template Specification Language

A *template* is a pre-defined form with parameters that are specified by either the user or the application at run-time. In YAG, each template is composed of two main parts: template slots and template rules. *Template slots* are parameters or variables that applications or users can fill with values. *Template rules* express how to realize a surface constituent. Templates are realized as strings by replacing slots in each rule with values from the application and then evaluating the rule. YAG template rules support nested and recursive templates. There are ten types of template rules.

The String rule returns a pre-defined string as a result.

The Evaluation rule evaluates the value of a template slot. If the value of the slot is another feature structure, then that structure is evaluated recursively. If the value of the specified slot is not a feature structure, this rule returns the value without any further processing.

The Template rule returns the result of instantiating a template with a given set of slot-value pairs.

The If rule is similar to an *if-then* statement in most programming languages, returning a result when the antecedent of the rule is true.

The Condition rule is similar to the *cond* statement in LISP, returning a result when one of its antecedent conditions is true.

The Insertion rule returns the result of interleaving the results of two template rules.

The Alternation rule selects one alternative template rule to be realized based on a uniform probability distribution, thereby adding variety into a generated text.

The Punctuation rule concatenates a punctuation mark to the specified end of adjacent strings. The position of a punctuation mark is either `left`, `right`, or `both`.

The Concatenation rule appends the the result of one template rule with the results of a second rule.

The Word rule is used in association with predefined functions and a lexicon to realize expressions that should not be "hard-coded" in a template, such as the inflected forms of a word from the dictionary or the cardinal/ordinal number corresponding to an integer.

Figure 1 shows the template rules that would be used to express propositions of the form `has-property(agent, pname, pval)`, such as `has-property(John, age, 20)`, which corresponds to *John's age is 20*). These rules are part of

```
((COND (IF (equal pname nil)
           (EVAL agent)
       )
       (IF (not (equal pname nil))
           ((CONCAT (EVAL agent)
                    (S "'s"))
            (EVAL pname))
       ))
 (TEMPLATE verb-form
           ((verb "be")
            (person (agent person))
            (number (agent number))
            (gender (agent gender))
            (tense present)) )
 (COND (IF (not (equal property nil))
           (EVAL property)
       )
       (IF (not (equal pval nil))
           (EVAL pval)
       ))
 (PUNC "." left)
)
```

Figure 1: Examples of Template Rules

the OBJECT-PROPERTY semantic template. The rules use the template slots **agent**, **pname**, **pval**, and **property** and the template rule types IF, CONCAT, S, TEMPLATE, COND, EVAL, and PUNC. If **agent** = "John", **pname** = "age", and **pval** = "20", the surface text will be *"John's age is 20."*.

3 Examples of YAG in use

YAG provides facilities for generation from two types of inputs, a feature structure or a knowledge representation. The latter is accomplished by the use of a knowledge representation specific component that must be defined for the particular knowledge representation language to be used.

3.1 Generation from a Knowledge Representation Structure

Example 1, shows a knowledge representation input to YAG.[1] It contains two propositions and a list of control features. In this representation, M2 is the proposition that the discourse entity B2 is a member of class *"dog"*. M5 is the proposition that the name of the discourse entity B2 is *"Pluto"*. Thus, we can read the whole proposition as *"Pluto is a member of class dog."* or simply *"Pluto is a dog."*. The control features state that the output should be generated as a declarative sentence with "be" as the main verb.

Example 1 *Pluto is a dog.*

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

When processing this input, YAG treats the first proposition as the primary proposition to be realized. YAG will map the MEMBER-CLASS proposition to the template shown in Figure 2.

The control features, **form** = decl and **attitude** = be, are also used in selecting the template. (If the form had been interrogative, a template for generating a yes-no question would have been used.)

Example 2 shows an example where prominalization is specified as part of the control features. The primary proposition says that the agent (B4) is doing the action *"take"* on the object (B6). This proposition, along with the selected control features (**form** = decl and **attitude** = action), allows YAG to select the clause template.

```
((EVAL member)
 (TEMPLATE verb-form
           ((process "be")
            (person (member person))
            (number (member number))
            (gender (member gender))) )
 (EVAL class)
 (PUNC "." left) )
```

Figure 2: A member-class Template.

Example 2 *"He takes it."*

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

To override the **gender** default (NEUTRAL) of B4 and generate *"He"* instead of *"It"*, Example 2 specifies B4's **gender** as MASCULINE. To override the default expression type (full noun phrase) for both B4 and B6, Example 2 specifies (pronominal YES (B6 B4)) which forces pronominalization.

3.2 Generation from a Feature Structure

Example 3 shows a complete feature structure that would be used to realize the text *"Blood pressure involves your heart and blood vessels."*. Within a feature structure, the name of the template that YAG should use is given by the *template* feature. Thus, in this example, YAG retrieves the clause template[2] which is shown in Figure 3.

In the clause template, the **agent** slot is bound to *"blood pressure"* since its value is another feature structure representing the noun-phrase template. The **Evaluation** rule then realizes it as *"blood pressure"*. The **Template** rule realizes the verb *"involves"*, by evaluating the verb-form template with the **process** value taken from the clause template. The other slots (which would normally be taken from

[1] The knowledge representation language used in these examples follows the definition of SNePS case frames described in (Shapiro et al., 1996). SNePS is a semantic network processing system (Shapiro and Rapaport, 1992). However, inputs to YAG are parenthesized lists of symbols, not SNePS data structures.

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

Example 3 *"Blood pressure involves your heart and blood vessels."*

```
((TEMPLATE CLAUSE)
 (PROCESS "involve")
 (AGENT
     ((TEMPLATE NOUN-PHRASE)
      (HEAD "blood pressure")
      (DEFINITE NOART)))
 (AFFECTED
     ((TEMPLATE NOUN-PHRASE)
      (HEAD ((TEMPLATE CONJUNCTION)
             (SENTENCE NO)
             (FIRST ((TEMPLATE NOUN-PHRASE)
                     (HEAD "heart")
                     (DEFINITE NOART)))
             (SECOND ((TEMPLATE NOUN-PHRASE)
                      (HEAD "blood vessel")
                      (NUMBER PLURAL)
                      (DEFINITE NOART)))))
       (POSSESSOR ((TEMPLATE PRONOUN)
                   (PERSON SECOND))))))
```

the **agent** slot, if its value were available) are filled by defaults (the defaults for **number**, **person**, and **gender** are `SINGULAR`, `THIRD`, and `NEUTRAL`, respectively.) within the `verb-form` template. The next **Evaluation** rule realizes *"your heart and blood vessels"*, which is the result of realizing the **affected** slot (its value is a feature structure representing the `noun-phrase` template). Finally, the surface string is concatenated with a punctuation ".".

```
((EVAL agent)
 (TEMPLATE verb-form
           ((process ^process)
            (person (agent person))
            (number (agent number))
            (gender (agent gender))) )
 (EVAL affected)
 (PUNC "." left) )
```

Figure 3: A simplified template rule of the `clause` template.

4 Conclusion

We have presented a natural language generation component, called **YAG** (**Y**et **A**nother **G**enerator), that has been designed to meet the needs of real-time, interactive systems. YAG combines a fast, template-based approach for the representation of text structures with knowledge-based methods for representing content. Its inputs can include concepts or propositions along with optional annotations to specify syntactic constraints. YAG can also realize text from a feature-based representation of syntactic structure. YAG can detect and correct missing or conflicting features by using a preprocessor based on attribute grammars. (One can also specify default values in the grammar itself.) YAG uses an expressive, declarative language for specifying a generation grammar. The YAG distribution includes a graphical tool for extending and testing templates. In these ways, YAG provides the speed, robustness, flexibility, and maintainability needed by real-time natural language dialog systems.

References

Songsak Channarukul, Susan W. McRoy, and Syed S. Ali. 2000. Enriching Partially-Specified Representations for Text Realization. In *Proceedings of The First International Natural Language Generation Conference*, Israel.

Songsak Channarukul. 1999. YAG: A Natural Language Generator for Real-Time Systems. Master's thesis, University of Wisconsin-Milwaukee, December.

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

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

Barbara J. Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber. 1986. *Readings in Natural Language Processing.* Morgan Kaufmann Publishers, Los Altos, CA.

William C. Mann. 1983. An overview of the Penman text generation system. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, pages 261–265, Washington, DC, August 22-26,. Also appears as USC/Information Sciences Institute Tech Report RR-83-114.

Kathleen R. McKeown. 1982. The TEXT system for natural language generation : An overview. In *Proceedings of the 20th Annual Meeting of the ACL*, pages 113–120, University of Toronto, Ontario, Canada, June 16-18,.

Kathleen R. McKeown. 1985. Discourse strategies for generating natural-language text. *Artificial Intelligence*, 27(1):1–42. Also appears in (Grosz et al., 1986), pages 479-499.

Stuart C. Shapiro and William J. Rapaport. 1992. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5).

Stuart C. Shapiro, William J. Rapaport, Sung-Hye Cho, J. Choi, E. Feit, Susan Haller, J. Kankiewicz, and Deepak Kumar, 1996. *A Dictionary of SNePS Case Frames.* Department of Computer Science, SUNY at Buffalo.