Text Realization for Dialog *

Susan W. McRoy Songsak Channarukul Syed S. Ali
mcroy@uwm.edu, songsak@uwm.edu, syali@uwm.edu
Natural Language and Knowledge Representation Research Group
Electrical Engineering and Computer Science Department
University of Wisconsin-Milwaukee

Abstract

We present a natural language realization component, called YAG, that is suitable for intelligent tutoring systems that use dialog. Dialog imposes unique requirements on a generation component, namely: dialog systems must interact in real-time; they must be capable of producing fragmentary output; and they may be re-deployed in a number of different domains. Our approach to real-time natural language realization combines a declarative, template-based approach for the representation of text structure with knowledge-based methods for representing semantic content. Possible text structures are defined in a declarative language that is easy to understand, maintain, and reuse. A dialog system can use YAG to realize text structures by specifying a template and content from its knowledge base. Content can be specified in one of two ways: (1) as a sequence of propositions along with some control features; or (2) as a set of feature-value pairs. YAG's template realization algorithm realizes text without any search (in contrast to systems that must find rules that unify with a feature structure).

Introduction

Designers of intelligent tutoring systems who might wish to include dynamically generated text often face a number of barriers between theoretical work in natural language generation and its implementation. The value of dynamically generated text is that one can adapt the output to the interactive context without having to write and store all possible output strings ahead of time; designers' concerns include: (1) Is there an existing generation system that the application can use? (2) How hard will it be to link the application to the generator? (3) Will the generator be fast enough? (4) How much linguistic information does the application need to provide in order to get reasonable quality output? (5) How much effort must the development team devote to writing a generation grammar that covers all the potential outputs of the application?

YAG (**Y**et **A**nother **G**enerator) is a real-time, general-purpose, template-based generation system that will enable interactive applications, such as tutoring systems, to add dynamically generated natural language output, without requiring that the application include extensive knowledge of the grammar of the target language. YAG supports knowledge-based systems by accepting two types of inputs: the application can either provide a feature structure (a set of feature value pairs) or provide a syntactically under-specified semantic structure that YAG will map onto a feature-based representation for realization. YAG also provides an opportunity for the 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.

To use YAG, an application specifies the content to be generated, using either a feature structure or a set of propositions in a knowledge representation language. YAG then maps the components of these inputs onto the parameters of a template. (For a feature structure, the application selects the template; for a proposition, YAG will use the template that the application designer has previously indicated should be used whenever the given knowledge representation structure is used.) Finally, YAG recursively evaluates the instantiated template to produce a string of text. (Examples of these structures and the mechanism for generation will be described in greater detail later in this paper.)

YAG offers several benefits to Intelligent Tutoring System (ITS) designers:

Speed YAG has been designed to work in real-time. The YAG template processing engine does not need to use search to realize text, because an input specifies what templates should be used

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

and template execution never requires backtracking. Therefore 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; 1993; Mann 1983; McKeown 1982; 1985).)

Robustness In YAG, the quality of text is dependent on the quality of templates and also the completeness of input. However, 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, McRoy, & Ali 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. 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:

- 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 also addresses many of the practical barriers that prevent application builders from considering a new approach that comes from the research community: The YAG distribution includes an installation program and all the software needed to run YAG and some of the applications that already use it. (This includes Tcl/Tk, CLISP, and SNePS.) YAG 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. The YAG distribution also includes a user manual and an installation guide.

Related Work

The most similar work to YAG is *TG/2* (Busemann 1996; Busemann & Horacek 1998). TG/2 is a template-based generation system that employs a rule-based technique to control the realization of templates. Both YAG's templates and TG/2's rules allow specifications of canned text, templates, and syntactic representations. However, TG/2, like most rule-based approaches, allows that there might be more than one rule for the same category. Some of them might fail while executing, which leads to backtracking. In addition, templates in TG/2 follow traditional template-based approaches with a couple extensions (*i.e.,* calling other rules and executing external functions). YAG implements more a powerful template-based approach by adding several additional extensions. It is also more declarative, yielding higher maintainability and comprehensibility. Although it is possible to define a grammar using TG/2's rules, its current distribution does not attempt to do so. Most generated texts are canned with little grammatical information. In contrast, YAG provides a general-purpose template-based grammar that can be reused across domains.

FUF/SURGE (Elhadad 1992; 1993) uses a functional unification grammar to realize text. The realization process unifies a given input with the grammar, and generates text from the unification result. SURGE is an extensive grammar of English that is provided with the system. Other systems, such as *Penman/Nigel* (Mann 1983), do not use grammar unification but traverse its (systemic) grammar to generate texts are also time-consuming. Despite their ability to produce high quality text, the critical drawback is that their speed is dependent on the size of the grammar rather than the intended output.

Ideally, generation systems should be able to produce texts from different levels of abstraction, not only at the syntactic, but also at the semantic level.

They should also allow fragmentary output that might not conform to standard grammar rules but is valid in an application's domain. Templates serve well in this area because arbitrary text structure can be defined. By contrast, systems that enforce a particular grammatical theory (for example, *MUMBLE* (Meteer *et al.* 1987) and *RealPro* (Lavoie & Rambow 1997)) are less flexible in this regard. In YAG (and other template-based systems) templates can be defined liberally to suit the application needs. Additionally, YAG can generate texts incrementally, from a single word or a sentence, to multiple sentences.

In the remainder of this paper, we will describe YAG's template specification language, YAG's processing architecture, and some examples that illustrate the use of YAG from an application.

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.

The syntax of YAG's template rules is similar to the syntax for function expressions used in the programming language LISP (although the range of legal expressions is much smaller). There are ten types of template rules.

The String rule returns a pre-defined string as a result. The syntax of this rule is `(S` *string*`)`, where *string* is a list of characters enclosed in double quotes.

The Evaluation rule evaluates the value of a template slot. The syntax of this rule is `(EVAL` *slot-name*`)`, where *slot-name* is a symbol that names a slot in the template. 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 syntax of this rule is `(TEMPLATE` *template-name slot-list*`)`, where *template-name* is a symbol naming a template, *slot-list* is a list of the form `(` *slot-pair** `)`, and *slot-pair* is a two-element list in which the first element is a symbol naming a slot and the second element is either a string, a slot-name from the current template (marked with a preceding "^"), or a two element list containing two slot-names, one from the current template and one to be evaluated based on the value of the named slot from the current template.

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 syntax of this rule is `(IF` *template-condition template-expression* `[ELSE` *template-expression*`])` where *template-condition* is a boolean-valued condition on the value of a slot in the template and *template-expression* is either a single template rule or a list where each element is a template-rule. The first *template-expression* is evaluated when the condition is satisfied, otherwise the second one is evaluated. The `ELSE` part is optional.

The Condition rule is similar to the *cond* statement in LISP, returning a result when one of its antecedent conditions is true. The syntax of this rule is `(COND` *if-rule**`)` where each *if-rule* is an If template rule (without the `ELSE` part). The first if-rule that succeeds will be evaluated; the rest are ignored.

The Insertion rule returns the result of interleaving the results of two template rules. The syntax of this rule is `(INSERT` *template-rule template-rule integer*`)`. The value is the result of inserting the string corresponding to the first template rule into the string from the second template rule, at the position specified by the integer.

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 syntax of this rule is `(ALT` *template-rule**`)`.

The Punctuation rule concatenates a punctuation mark to the specified end of adjacent strings. The syntax of this rule is `(PUNC` *punc-string punc-position*`)`, where *punc-string* is a punctuation mark surrounded by double quotes and *punc-position* is either `left`, `right`, or `both`. The position `left` is used for punctuation to be attached at the end of a string, such as a period (".") or a question mark ("?"). The position `right` is used for punctuation to be attached at the front of a string, such as a left parenthesis ("("). The position `both` is used when the punctuation that needs to attach to both of its adjacent strings (*e.g.*, the colon (":") in `11:05 am.`).

The Concatenation rule appends the the result of one template rule with the results of a second rule. The syntax of this rule is `(CONCAT`

template-rule template-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. The syntax of this rule is (WORD *template-function template-parameters**).

With these rules implemented, templates are intrinsically declarative and powerful. They can reuse other relatively small templates to realize more complicated texts. In addition, general-purpose templates (comprising a general-purpose, template-based generation grammar) can be defined to realize domain-independent text thus increasing the reusability and portability of templates. The detailed implementation of template rules is given in (Channarukul 1999).

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 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."*.

```
((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

YAG's Architecture

Our implementation of YAG has a layered architecture as shown in Figure 2. This architecture allows an application to realize texts from two kinds of input, a knowledge representation or a feature structure. In addition, we separate the knowledge sources used by YAG from the "Core YAG", which includes the processes that retrieve and evaluate templates. The knowledge sources include a specification for mapping expressions from the knowledge representation language onto an associated template, a collection of templates that have been organized into libraries, and a lexicon.

The Core YAG

The **Knowledge Representation Realizer** realizes a knowledge representation into an appropriate feature structure. Its input contains two parts: a semantic network that represents content, and a set of control features that provides 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 **Feature Structure Realizer** realizes a feature structure into a surface text. 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 **Template Manager** stores a template into a specified library, and retrieves template structures for the Knowledge Representation Realizer and the Feature Structure Realizer. If a template has multiple names, these names (template *aliases*) will be maintained by the Template Manager.

The Knowledge Base

There are two types of knowledge bases in YAG; *domain dependent* and *domain independent*. The domain dependent knowledge base includes the *Mapping Table* and *Domain Template Libraries*. The domain independent knowledge base includes the *Syntactic Template Library* and the *Lexicon*.

The **Mapping Table** is used to map a knowledge representation to its associated template. The Knowledge Representation Realizer accesses a mapping table with selected control features to pick an appropriate template when realizing a text from a knowledge representation. Each mapping entry provides a declarative specification for constructing a feature structure from the propositions and control features.

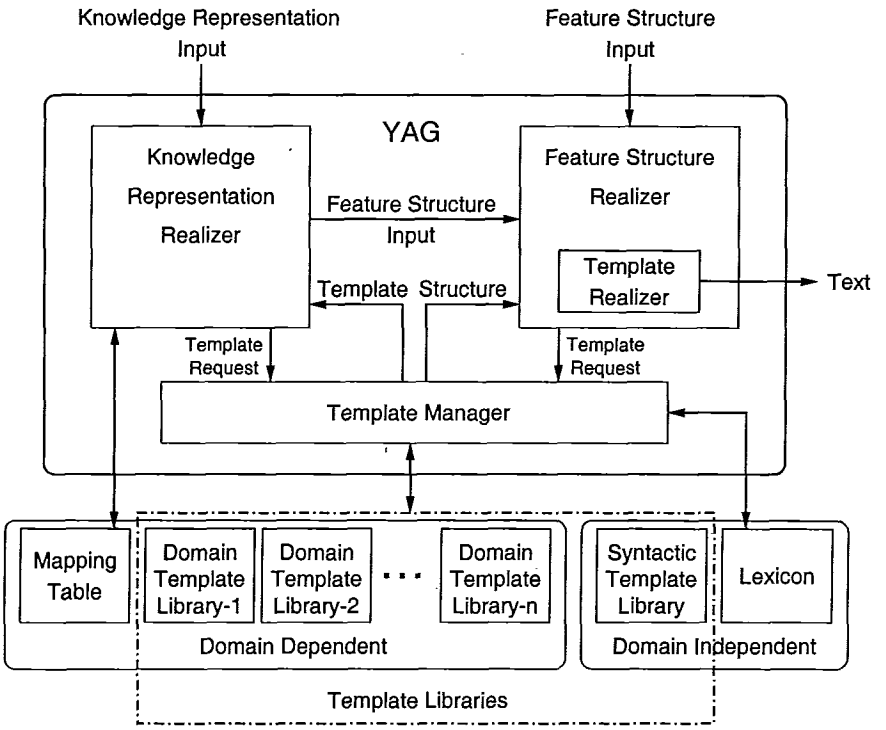

Figure 2: Architecture of YAG

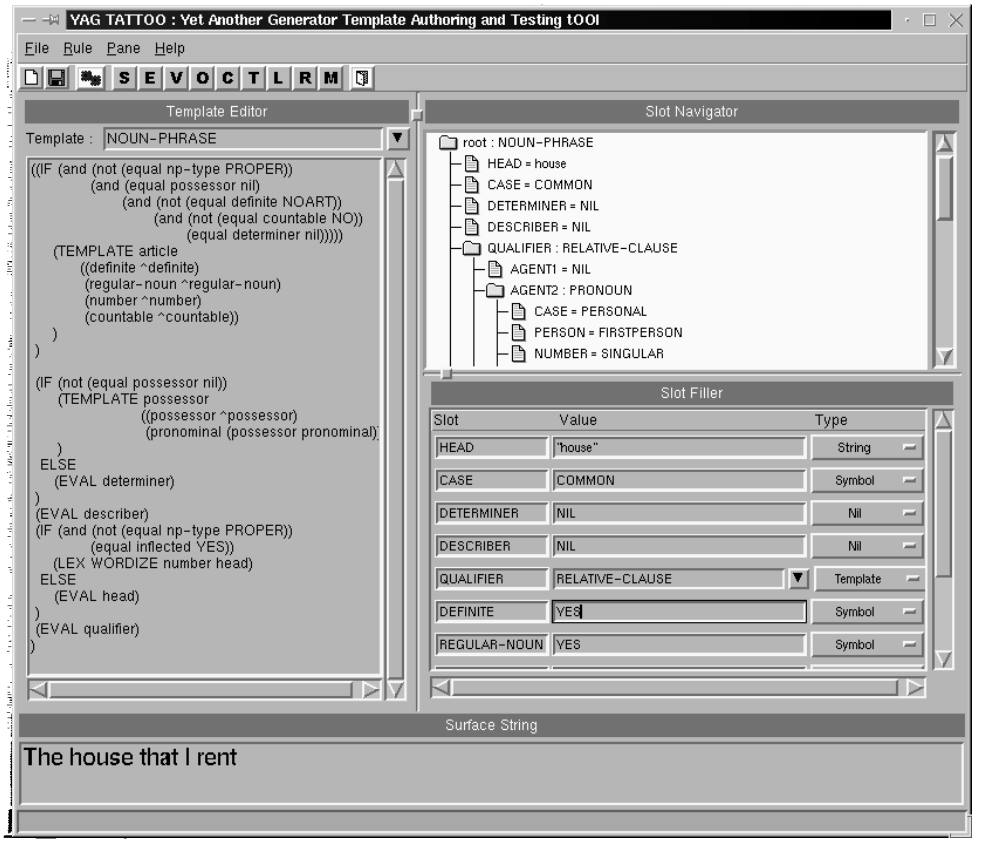

Figure 3: A screenshot of TATTOO.

Creating the mapping table is the primary task in constructing a new knowledge representation realization component for a given knowledge representation framework.

The **Domain Template Libraries** contain templates that are specific to a particular application. Developers can author their own templates when necessary by manually editing a text file that contains templates in any text editors, or by using **TATTOO** (The **T**emplate **A**uthoring and **T**esting **O**ol) which is a program that allows a knowledge engineer to author and test templates in a graphical environment. Its screenshot is shown in Figure 3.

The **Syntactic Template Library** contains templates that are used as a grammar of English, such as the CLAUSE, NOUN-PHRASE, and PRONOUN templates. Other templates can embed these templates to form more complex structures. They can also be combined with templates from the Domain Template Library.

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

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 (the mapping table) that must be defined for the particular knowledge representation language to be used.

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.

[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 & Rapaport 1992). However, inputs to YAG are parenthesized lists of symbols, not SNePS data structures.

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 4. 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.)

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

Figure 4: A MEMBER-CLASS Template.

Example 2 shows an example where prominalization is specified as part of the control features. The primary proposition says that there is the agent (B4) who 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.

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 spec-

ifies 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.

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 will use is given by the *template* feature. Thus, in this example, YAG retrieves the clause template[2] which is shown in Figure 5.

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))))))
```

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 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 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 as *"your heart and blood vessels"*, which is a 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 ".".

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

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

Figure 5: A simplified template rule of the CLAUSE template.

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 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. Through its speed, flexibility, and maintainability, YAG supports the needs of real-time natural language dialog systems.

References

Busemann, S., and Horacek, H. 1998. A flexible shallow approach to text generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, 238–247.

Busemann, S. 1996. Best-first surface realization. In Scott, D., ed., *Proceedings of the Eighth International Workshop on Natural Language Generation*, 101–110.

Channarukul, S.; McRoy, S. W.; and Ali, S. S. 2000. Enriching Partially-Specified Representations for Text Realization using An Attribute Grammar. In *Proceedings of The First International Natural Language Generation Conference.*

Channarukul, S. 1999. YAG: A Natural Language Generator for Real-Time Systems. Master's thesis, 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.

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. Reprinted in *Readings in Natural Language Processing*, (1986) Barbara Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber *eds.*, Morgan Kaufmann, Los Altos. 479-499.

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 Rapaport, W. J. 1992. The SNePS family. *Computers & Mathematics with Applications* 23(2-5).

Shapiro, S. C.; Rapaport, W. J.; Cho, S.-H.; Choi, J.; Feit, E.; Haller, S.; Kankiewicz, J.; and Kumar, D. 1996. *A Dictionary of SNePS Case Frames.* Department of Computer Science, SUNY at Buffalo.