Natural Language Engineering **9** (4): 381–420.
DOI: 10.1017/S1351324903003188 Printed in the United Kingdom

An augmented template-based approach to text realization

SUSAN W. MCROY, SONGSAK CHANNARUKUL and SYED S. ALI

Natural Language and Knowledge Representation Research Group, Electrical Engineering and Computer Science Department, University of Wisconsin-Milwaukee, MI, USA
e-mail: mcroy@uwm.edu

(*Received 29 January 2002; revised 30 October 2002*)

Abstract

We present an *Augmented Template-Based* approach to text realization that addresses the requirements of real-time, interactive systems such as a dialog system or an intelligent tutoring system. Template-based approaches are easier to implement and use than traditional approaches to text realization. They can also generate texts more quickly. However traditional template-based approaches with rigid templates are inflexible and difficult to reuse. Our approach *augments* traditional template-based approaches by adding several types of declarative control expressions and an attribute grammar-based mechanism for processing missing or inconsistent slot fillers. Therefore, augmented templates can be made more general than traditional ones, yielding templates that are more flexible and reusable across applications.

1 Introduction

This paper presents an *Augmented Template-Based* approach to text realization that addresses the requirements of real-time, interactive systems such as a dialog system or an intelligent tutoring system. Dynamically generated text is valuable in such applications because output can be adapted to the interactive context in real-time without system designers having to anticipate and store all possible output strings ahead of time.

Although many natural language generation systems are currently available (Hovy 1997; Reiter and Dale 1997), most of these systems are based on so-called full natural language generation techniques which are strongly motivated by theoretical linguistic studies. In such approaches, speed of realization depends on the overall size of the generation grammar, because the system must search the entire grammar each time that a sentence is generated. These systems are thus best suited to off-line tasks that require very high quality text, such as text summarization or document generation, where timing is not critical. These approaches also require that an application embed significant amounts of linguistic knowledge, which may be difficult if the application was not originally designed to produce natural language output.

A few natural language generation systems make use of a faster, shallower method called templates. A system that uses a template-based approach creates texts by filling in the slots of a template with values provided by the application, typically with little or no syntactic processing. This method is appropriate for generating a large number of similar documents that vary only on a few fields, as in mail-merge programs. Some template-based systems also allow simple syntactic processing such as pluralization of nouns, subject-verb agreement, and referring expressions, but require developers to modify a system's source code whenever they need to make changes. Thus the traditional approaches to template-based generation have been criticized for generating low quality texts and for being difficult to maintain (Reiter 1995).

Our approach to realization overcomes the inflexibility of traditional templates by augmenting them with several types of declarative control expressions. Fine-grained syntactic processing (if desired) can be done by defining the grammar of a text generator as a set of declarative templates. For example, we have built a syntactic grammar for YAG that is based partially on a Phrase Structure Grammar (PSG) (Smith 1991) and a Functional Grammar (Halliday 1994; Kay 1979).[1] To support grammar building, template specifications can refer to other template definitions, as needed. In addition, we provide an attribute-grammar based feature-structure enricher to assign default values to slots that depend on the values of other slots or to resolve conflicts between the values assigned to dependent slots. Thus, augmented templates can realize high quality texts in real-time, while remaining compact and reusable.

We have designed and implemented a system called **YAG** (**Y**et **A**nother **G**enerator) based on the use of augmented templates. YAG offers the following benefits to applications and application designers:

Speed: YAG has been designed to work in real-time. YAG is fast because:

- The YAG template processing engine does not use search to realize text. Slots in the input feature structure are used to retrieve (via hashing) the one realization template that is appropriate. The speed of generation of a sentence thus depends upon the size of the template that is used to generate that sentence and the number of embedded templates that it contains, rather than on the total number of rules or templates in the system's generation grammar. (We consider the speed of a number of test cases in section 3.5.)
- The template authoring tool for YAG (as well as YAG's template processing engine) enforces the restriction that any given input has only one appropriate realization template. Prior to the realization of a semantic structure, the designer will have designated the most appropriate top-level template for that type of structure. An application can also select a template on the fly, if it specifies the input to be realized by providing a feature structure.

[1] This grammar is described in Channarukul (1999), and is included with the YAG distribution that we make available to other researchers.

- YAG is deterministic. Unless a template makes explicit use of random alternation (described in section 2.1), the realized text for a given input is always the same. Also, because there is no search, there is never any backtracking.

Robustness: In YAG, the realization of a template cannot fail. Inputs that cause other generators to fail, such as cases of subject-verb disagreement, are handled according to the preferences of the application. YAG can either produce ungrammatical output to match the input, or for applications that need to enforce grammaticality, it can use the YAG feature-structure enricher to detect missing or conflicting features and supply acceptable values. The enricher makes use of a declarative specification of slot constraints, based on an attribute grammar (Channarukul, McRoy and Ali 2000).[2] This specification is modifiable and extensible by the application designer.

Expressiveness: YAG offers an expressive language for specifying a generation grammar and general templates alike. 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 upon 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 re-use 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.

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

[2] When the feature structure enricher is used, a failure can occur, if there are circular dependencies in the attribute grammar, but this rarely occurs.

a statement, as a noun-phrase rather than a sentence, or as a pronoun rather than a full noun phrase.

We note that YAG is a realization approach and system. It, by itself, does not include any theory of discourse or inter-sentential anaphora – although any application that makes use of such a theory is free to do so using YAG. Anaphora within a sentence can be handled either by the text planning application, or within YAG using templates or attribute grammar rules that capture the dependencies directly.[3] At present, we have only used YAG to realize single sentences. Thus, inter-sentential constraints must be handled by the application. In theory, multiple sentence templates could be devised to deal with multi-sentential constraints.

The following sections explain our approach to augmented templates, overview the architecture of an implemented system that uses this approach, presents a detailed example of realization from a semantic representation, and describes other related work.

2 An augmented template-based approach

In our approach to realization, templates are defined by a knowledge expert to capture the range of syntactic structures that are needed. The knowledge expert must also provide a mapping between the type of inputs provided by the application (such as semantic networks) and the names used in YAG feature structures. Then, during run-time, an application can realize text by providing YAG with a description of the content to be realized (in its internal format), along with control parameters that are used to help select and instantiate a template. YAG will map these inputs onto a feature structure, possibly revising this structure to correct missing or inconsistent values, then instantiate the selected template, and finally interpret the instantiated template to produce a text. The power of YAG derives from the expressiveness of its language for defining templates and from its ability to recover from omissions and errors in its input.

This section describes our augmented template-based approach in more detail. First, we describe how templates are defined. Then, we discuss how templates are instantiated and realized with values provided by an application. Finally we present our approach to enhancing template values to enable the realizer to produce text for inputs that have been only partially specified by the application. In section 5, we consider our implementation of this approach and trace the realization of an end-to-end example.

2.1 The definition of templates

In our template-based approach, each template is composed of two main parts: template slots and template rules. *Template slots* are parameters or variables that users or applications can fill with values. In a template definition, one must provide the name of the slot and a default value, if any. (We also provide a separate

[3] Of the two approaches supported by YAG, adding attribute grammar rules is the easiest for both the grammar writer and the application.

mechanism, based on attribute grammar, for specifying values for missing slots. See section 2.3.2.) *Template rules* are declarative statements that define how inputs to the template should be realized as text. When a template is realized, YAG interprets each template rule in the order it was defined, using the values from the input that have been mapped to particular template slots.

Template rules can be used to specify simple combinations of text or a hierarchically defined, linguistically based grammar. Our template specification language consists of the following types of rules:

- The **String** rule returns a pre-defined string as a result, allowing an application to included canned text.
- The **Evaluation** rule evaluates the value of the specified slot, which can be either a feature structure or a value. Evaluation is recursive, allowing one to embed templates within a feature structure to express complex texts. If the value of the specified slot is not a feature structure, this rule returns the value of the slot without any further processing.
- The **Template** rule specifies a template name with a set of slot-value pairs for realization. This allows one to embed template definitions within each other, so that common templates, such as noun phrases, can be reused.
- The **If** rule is a conditional expression, similar to the *if-then* statement in most programming languages. For example, the condition may test whether the value of a slot is equal to some constant. The rule will realize a text only if the condition is satisfied. This rule is typically used with the Condition rule, described below.
- The **Condition** rule is similar to the *cond* statement in the Lisp programming language. The condition rule contains a sequence of If rules. During interpretation, the conditions will be evaluated until the first one is satisfied and the corresponding value will be realized. The rest will be ignored.
- The **Insertion** rule manipulates the generated strings (from other rules). This is useful when the order of words in the surface string is different from the order given in a template. For example, suppose that the first rule generates the word *"we"* and the second rule generates *"can do"*. When generating a question, we can invert the order of the pronoun and the auxiliary by inserting the result of the first rule into that of the second rule, thus forming a new text *"can we do"*.
- The **Alternation** rule specifies a set of alternative template rules for realizing a text. During interpretation, YAG will randomly select a single alternative (with uniform probability) for realization. This allows for a limited amount of non-determinism in a generally deterministic system. It is useful for introducing a bit of variety into realized texts, where there is no conditional expression that could guide the system to favor one alternative over another.
- The **Punctuation** rule allows the concatenation of a punctuation mark to its adjacent strings. The rule can specify whether the punctuation should be attached on the *left*, on the *right*, or to *both* adjacent strings. Most punctuation is at the end of a string (left punctuation), such as a period (“.”), a semicolon (“;”), and

a question mark ("?"). However, in some cases, a punctuation is in front of a string (right punctuation). Examples are a left parenthesis ("(") and a dollar sign ("$"). A punctuation mark that needs to attach to both of its adjacent strings is, for example, a colon (":") in the formatting of time (e.g. `11:05 am.`).

- The **Concatenation** rule concatenates surface strings of two rules without a space (a blank character) between them.
- The **Word** rule is used in association with pre-defined functions and a lexicon (or dictionary) to realize a word that cannot be generated by template rules. The inflection of nouns and verbs is a typical situation where this kind of template rule is appropriate, since we do not want to store all forms of nouns and verbs in templates. In addition, the generation of some kind of strings is beyond the capability of general template rules. For example, to generate a string *"twenty-four"* from a given numerical value *"24"*, we need a procedural definition because it is not possible to define a declarative template that covers all numbers.

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 re-usability and portability of templates. The speed of realization depends on the complexity of templates – complex templates will take longer to realize, but not the overall size of the template grammar. The implementation of template rules is described in detail in Channarukul (1999).

2.2 The instantiation and realization of templates

During template instantiation, template slots are given values from the input. After instantiation, text will be realized by interpreting each template rule, defined as described in the previous section, in the order in which it appears.

Templates can be instantiated with values provided in either of two forms. Values can either be provided as a feature structure or as a sequence of propositions (from a knowledge base). A feature structure specifies the template to be instantiated, the content to be generated, and any important syntactic constraints. The names of features must be identical to slots in the selected template. A sequence of propositions specifies the content to be generated along with control parameters that are needed to select a template (such as whether the content should be expressed as a question or a statement). If propositions are used as input, then the system also requires a separate specification of the mapping from each type of proposition to appropriate templates. The mapping will specify how arguments within a proposition are to be mapped to template slots. This mapping is specified once and used for the instantiation of all templates for the application. Different mappings can be created to provide customized text for different domains or applications.

We will now consider some examples of both types of input formats and the resulting realized texts. For these examples, we will assume that the input specifies most, but not all of the slots needed by the templates. Missing values will be taken from defaults, given in the definition of a template as part of the specification of

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

Fig. 1. Simplified template rule of the clause template.

template slots. (A list of most defaults is given later in Table 1.) However, our approach also supports a second, more flexible mechanism for handling partially specified inputs that incorporates a separate processing step with an attribute grammar. We will discuss both mechanisms in greater detail in section 2.3.

2.2.1 Natural language generation from feature structures

In YAG, the most direct form of input is as a feature structure. Feature structures are composed of one or more features and their values. Within a feature structure, the name of the template is specified in the **template** feature. As mentioned above, the names of features must correspond to slots in a template. Values for a feature can be a string, a symbol, or an embedded feature structure.

Example 2.1 shows a feature structure that could be used to realize into the text *"John walks."*. This structure selects the clause template and provides values for two slots: the **agent** should be the string, "*John*" and the **process** should be "walk".

Example 2.1
"John walks."

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

For this example, we assume that the template has been defined using the template rules that are shown in figure 1.[4] This definition says that a clause is to be realized as the value of the **agent**, followed by a verb-form (which is defined as an embedded template, where the ^ symbol indicates that the template should use the **process** slot from the enclosing clause template[5]), followed by the **object**, with a period concatenated to its left end.

Thus, during realization, the rule `(EVAL agent)` is realized first, yielding *"John"*. Then, the template rule that will realize the verb-form template is processed. In this rule, the value already bound to the **process** slot in the input structure i.e. "*walk"* is

[4] These rules have been simplified to facilitate explanation.
[5] The **person**, **number**, and **gender** slots are to be taken from the values of the features within the **agent** slot, if it has been defined as a feature structure. If the agent is not a feature structure, then default values must be used.

passed to the verb-form template. The other parameters of the verb-form template (**person**, **number**, and **gender**) expect values from the **agent** slot, but because it is a string, not a feature structure, default values (which are `THIRD`, `SINGULAR`, and `NEUTRAL`, respectively) are used. This results in the generated verb *"walks"*. The partial surface string is *"John walks"*.

Since no value is given to the **object** slot and it has no default, the rule `(EVAL object)` returns nothing. Finally, the surface string is concatenated with a punctuation ".", yielding the fina l surface string *"John walks."*.

Example 2.2
Feature structure input for "*He understands it*."

```
((template clause)
 (process "understand")
 (agent ((template noun-phrase)
         (np-type PROPER)
         (head "George")
         (gender MASCULINE)
         (pronominal YES)))
 (object ((template noun-phrase)
          (head "book")
          (pronominal YES))) )
```

Example 2.2 shows a feature structure from which the string *"He understands it."* would be generated. This example illustrates that the value of any feature may be another feature structure, as is the case for the **agent** and **object** slots. It also discusses how an application may request that a pronoun be used. Allowing a feature structure to embed another feature structure increases the flexibility of a template-based approach.

In this example, the embedded feature structure

```
((template noun-phrase)
 (np-type PROPER)
 (head "George")
 (gender MASCULINE)
 (pronominal YES))
```

would be processed first, using the noun-phrase template. This would return the word *"he"*, because of **pronominal** and **gender** constraints. This in turn would be bound to the **agent** slot of the clause, and realized as the string *"he"*. When the verb-form slot is processed, the word "understands" is realized, using the same defaults as the previous example. Next, the value of the **object** slot is realized as the string *"it"*. Finally, the clause template returns the string *"He understands it."* as output.

Example 2.3
"Does blood pressure involve your heart and blood vessels?"

```
((template clause)
 (mood YES-NO)
 (process "involve")
 (agent ((template noun-phrase)
```

```
        (head "blood pressure")
        (definite NOART)))
 (object ((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 last example in this section, Example 2.3, presents a feature structure input that uses several embedded and nested templates. It is used to generate the sentence *"Does blood pressure involve your heart and blood vessels?"*. Yes-no type questions can be generated by specifying the **mood** slot as `YES-NO`, while the rest of feature structure remains the same as for a declarative clause. Noun phrases generally require one to specify the head and allow one to specify the definiteness of the noun phrase by making the article definite (`(DEFINITE YES)`), indefinite (`(DEFINITE NO)`), or omitting the article altogether (`(DEFINITE NOART)`) and a possessor. (The complete noun-phrase template is given later in figure 20.) In this example, the **possessor** slot uses the pronoun template to generate the pronoun *"you"*. However, within the noun-phrase template, the value of the **possessor** slot will, in turn, be used to instantiate an embedded template, possessive-pronoun. This template causes the word to be realized as *your*.[6]

In other respects, the realization of this example is similar to previous ones.

2.2.2 Natural language generation from knowledge representation

In addition to accepting inputs as feature structures, YAG also allows one to specify content using a sequence of propositions from a knowledge base, along with a few simple control features. This form of input decreases the amount of linguistic knowledge required by an application. YAG can be used with any knowledge representation language, by specifying a single knowledge representation specific component, called a mapping table, for the language used. This component will map the knowledge representation into a feature structure that can be processed by the same mechanisms used to process the feature structures described in section 2.2.1.

Generation from a knowledge representation is different than from feature structures in that the knowledge representation structures themselves need not contain any linguistic information. The control features allow the application to specify whether the content should be expressed as a statement, a question, or a

[6] YAG includes templates for each pronoun form (subjective, objective, possessive, or reflexive).

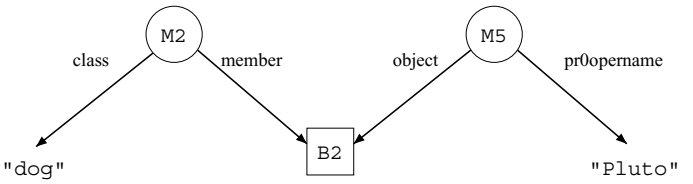

Fig. 2. Semantic network for the sentence *"Pluto is a dog."*

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

Fig. 3. Example semantic network description for YAG.

noun phrase. They also allow one to specify a mental attitude, such as knowing or wanting, or to override some other default provided by YAG, such as whether or not to use a pronoun to realize a noun phrase. The control features, along with the type of the primary relation (the first one in the sequence) are used to uniquely select a template. This selection, and the mapping between arguments and feature value pairs, is accomplished with the help of a mapping table, and specialized algorithms for combining information from different propositions. We will now consider a few examples of inputs provided as propositions in a knowledge representation language; however we will defer our discussion of *how* such inputs are realized until section 5, when we consider a detailed example from end to end.

The knowledge representation that is used in this paper is SNePS, which uses propositional semantic networks (Shapiro and Group 1998; Shapiro and Rapaport 1992). 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). 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 the `MEMBER` arc and the `CLASS` arc. In addition to the mapping table, we have also defined a number of domain specific templates for generating specific case frames, such as `MEMBER-CLASS`, `AGENT-ACT`, `ACTION-OBJECT`, and `OBJECT-PROPERTY`. Note that although we have defined a number of semantic templates, these templates reuse previously defined syntactic templates such as verb-form or noun-phrase.

Figure 2 shows an example of a propositional semantic network. Figure 3 shows this same example in the input format from SNePS that we use when interacting with YAG. In this 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 the class dog."* or simply *"Pluto is a dog."*.

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

Fig. 4. The Member-Class template.

Example 2.4 illustrates how we would provide this network as an input to YAG to have it realized as a simple declarative sentence.

Example 2.4
"Pluto is a dog."

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

YAG shall map the `MEMBER-CLASS` proposition to the template shown in Figure 4. The control features **form** = `decl` and **attitude** = `be`, have been used in selecting the exact template to be used. (The attitude "be" corresponds to a simple statement of fact.) If the **form** had been `interrogative`, a different template would have been used.

Example 2.5 shows the input used to realize "George reads the book." This example shows how control features can be used to override defaults made by YAG. By default, YAG would realize a common noun using an indefinite form (i.e. "a book"). To produce the definite noun phrase (*"the book"*), we must override the definiteness default for the discourse entity `B6`. This is done by adding the control feature: `(definite YES B6)`.

Example 2.5
"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) ))
```

Example 2.6 shows how control features can be used to realize the pronouns in the sentence "He understands it". In this example, the proposition says that there is an

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.

Example 2.6
"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) ))
```

To override the default expression type (full noun phrase) for both B4 and B6, Example 2.6 specifies `(pronominal YES (B6 B4))` which forces pronominalization on both discourse entities named. To override the **gender** default (`neutral`) of B4 and generate *"he"* instead of *"it"*, another control feature specifies that B4's **gender** is `MASCULINE`.

2.3 Partially-specified input

In general, a text realization system requires a great deal of syntactic information from an application to generate a high quality text; however, an application might not have this information (unless it has been built with text generation in mind). This problem has been referred to as the *Generation Gap* (Meteer 1990). Meteer first identified the generation gap problem as arising at the text planning stage. A *text planner* must decide what content needs to be expressed and creates a corresponding text plan for generating it. A *sentence planner* is then used to select an appropriate syntactic structure for a given plan. Typically, neither a text planner nor a sentence planner is concerned with fine-grained syntactic issues, such as whether the subject of the sentence is a singular or plural noun. Thus, it becomes the responsibility of a text realizer to infer the missing information and to generate the best possible text from a given input.

2.3.1 Defaulting mechanism

Most generation systems (such as FUF/SURGE (Elhadad 1992), Penman (Mann 1983), RealPro (Lavoie and Rambow 1997), and TG/2 (Busemann 1996)) handle the missing information in an input by using *defaulting*, in which a grammar writer specifies a default for each syntactic constraint. Similarly, YAG supports defaulting by allowing grammar writers to specify a default for each slot in templates. Another approach that has been proposed by others, such as Knight and Hatzivassiloglou (1995), is to fill in the missing information on the basis of word co-occurrence data

```
((template clause)
 (process-type MENTAL)
 (process "want")
 (processor ((template noun-phrase)
             (head ((template conjunction)
                    (first ((template noun-phrase)
                            (head "Jack")
                            (np-type PROPER)
                            (gender MASCULINE)
                            (definite NOART)))
                    (second ((template pronoun)) )))
             (person SECOND)
             (number PLURAL)) )
 (phenomenon ((template noun-phrase)
              (head "dog")
              (definite NOART)
              (possessor ((template noun-phrase)
                          (head "sister")
                          (gender FEMININE)
                          (definite NOART)
                          (possessor ((template noun-phrase)
                                      (head "Jack")
                                      (np-type PROPER)
                                      (gender MASCULINE)
                                      (pronominal YES)
                                      (definite NOART)) )))))
 (rear-circum ((template clause)
               (mood TO-INFINITIVE)
               (process-type MATERIAL)
               (process "swim")) ) )
```

Fig. 5. Feature structure for the sentence *"Jack and I want his sister's dog to swim."*

collected from a large corpus of text. However, this approach will have difficulty when there are long-distance dependencies among constituents in a text, because of the difficulty in collecting sufficient statistical information about longer patterns.

In YAG, values for the templates are provided by an application; inputs can include either a conceptual representation of content or a feature structure. When an input is only partially specified, defaults defined *a priori* in a template will be applied. Figure 5 shows an example of YAG's feature-structure based input, that would realize the sentence *"Jack and I want his sister's dog to swim."*. This input is partially specified and thus is more compact and easier for an application to specify, than a complete specification. Table 1 shows the features that have been omitted and the defaults used by YAG to realize the sentence from the input.

2.3.2 Using attribute grammars for defaulting

Even though simple defaulting is useful, it is still inflexible and prone to errors. This is because there might not be one default that suits all applications or situations. YAG makes use of an Attribute Grammar (Knuth 1968; Alblas 1991) to enrich

Table 1. *Some defaults from YAG's syntactic templates*

Template name	Template slot	Default	Allowed values
CLAUSE	sentence	YES	YES, NO
	mood	DECLARATIVE	DECLARATIVE, YES-NO, WH, IMPERATIVE, TO-INFINITIVE
	process-type	ASCRIPTIVE	ASCRIPTIVE, MENTAL, MATERIAL, COMPOSITE, POSSESSIVE, LOCATIVE, TEMPORAL, VERBAL, EXISTENTIAL
	mode	nil	ATTRIBUTIVE, EQUATIVE, CAUSATIVE
	tense	PRESENT	PRESENT, PAST
	future	NO	YES, NO
	progressive	NO	YES, NO
	perfective	NO	YES, NO
	voice	ACTIVE	ACTIVE, PASSIVE
	quality	POSITIVE	POSITIVE, NEGATIVE
NOUN-PHRASE	np-type	COMMON	COMMON, PROPER
	person	THIRD	FIRST, SECOND, THIRD
	number	SINGULAR	SINGULAR, PLURAL
	gender	NEUTRAL	NEUTRAL, MASCULINE, FEMININE
	definite	NO	YES, NO, NOART
	regular-noun	YES	YES, NO
	countable	YES	YES, NO
	inflected	YES	YES, NO
	pronominal	NO	YES, NO
POSSESSOR	pronominal	YES	YES, NO
PRONOUN	type	PERSONAL	PERSONAL, OBJECTIVE, REFLEXIVE, POSSESSIVE-PRONOUN, POSSESSIVE-DETERMINER, RELATIVE, DEMONSTRATIVE
	person	FIRST	FIRST, SECOND, THIRD
	number	SINGULAR	SINGULAR, PLURAL
	gender	NEUTRAL	NEUTRAL, MASCULINE, FEMININE
CONJUNCTION	sentence	NO	YES, NO

partially-specified inputs (inputs that would normally require simple defaulting –

```
((template clause)
 (process-type MENTAL)
 (process "want")
 (processor ((template conjunction)
             (first ((template noun-phrase)
                     (head "Jack")
                     (np-type PROPER)
                     (gender MASCULINE) ))
             (second ((template pronoun)) )))
 (phenomenon ((template noun-phrase)
              (head "dog")
              (possessor ((template noun-phrase)
                          (head "sister")
                          (gender FEMININE)
                          (possessor ((template noun-phrase)
                                      (head "Jack")
                                      (np-type PROPER)
                                      (gender MASCULINE)
                                      (pronominal YES)) )))))
 (rear-circum ((template clause)
               (mood TO-INFINITIVE)
               (process-type MATERIAL)
               (process "swim")) ) )
```

Fig. 6. A (shorter) feature structure of the sentence *"Jack and I want his sister's dog to swim."*

Consider the example previously shown in figure 5; although the input is already more compact than a full specification, further simplification of the input provided from an application would have been possible – if certain inferences could be made. For example, the input structure given in figure 6 could replace the one given in figure 5. In figure 6, it is not necessary for the application to specify that the conjunction of two noun phrases is a plural noun phrase, nor that component noun phrases (proper nouns, pronouns, and possessives) should not contain an article. In the case of conjunctions, there is no default that would provide the correct outputs in all cases. Instead of simple defaulting, our approach uses an attribute grammar to make the appropriate inferences and automatically enrich the feature structure input so that neither the application, nor the templates need to be altered to handle dependencies correctly.

To make this enrichment possible, we must first extend a grammar to capture the propagation semantics of attributes of a target language (here, US English). This extension involves defining attributes and associated semantic rules. Using this additional information, YAG's feature-structure enricher constructs a tree from a given input and retrieves semantic rules and attributes from associated templates.

There are two types of attributes: inherited – those propagated down the tree; and, synthesized – those propagated up the tree. Both types of attributes are used to provide missing information in the input by a process of attribute evaluation.

Attribute evaluation begins by instantiating each inherited attribute with values from the input and then the remaining attributes are evaluated. This process is incremental in the sense that new information gained from previous evaluations

might lead to the discovery of additional information. When all attributes remain unchanged, or there is a conflict detected in the input, the process terminates. The generator then passes the enriched input to the realization component. Defaults are applied thereafter on any slots that are left unfilled.

Consider the following fragment of input from figure 6 that uses the conjunction template to join a noun phrase and a pronoun:

```
((template conjunction)
 (first ((template noun-phrase)
         (head "Jack")
         (np-type PROPER)
         (gender MASCULINE) ))
 (second ((template pronoun)) ))
```

This fragment is the subject of the sentence, therefore features such as **person**, **number**, and **gender** would be required to enforce subject-verb agreement (of English). The dependencies are captured by the semantic rules for the conjunction template, the noun-phrase template, and the pronoun template, shown in Table 2. Below we will provide an English gloss of these rules; the complete syntax of these rules is described in Channarukul *et al.* (2000).

For the conjunction template, the grammar will:

- Use the **sentence** feature of the current template (which is `NO` by default).
- Pass up the **person** feature found by comparing the **person** features associated with the two conjuncts (i.e. pass up second person whenever the conjuncts combine either first person and second or third person, or they combine second person and third person; pass up third person if both conjuncts use third person; otherwise pass up `nil`);
- Constrain the **number** feature to be `PLURAL`, the **gender** feature to be `NEUTRAL`, the **definite** feature to be `NOART`, and the **sentence** feature to the same as the sentence feature of the conjuncts.

For the noun-phrase template, the grammar will

- Require this template to enforce the inherited values of the **definite**, **number**, and **np-type** features.
- Require that the (embedded) determiner template enforce the **number** feature of the current template.
- Pass up four features (**definite**, **number**, **person**, and **np-type**) to any templates that use this noun phrase, where the following constraints apply:

  The definiteness feature that is passed is `YES` whenever the current template has inherited `YES` for this value or there is a possessor or a determiner and one of them passes up `YES` for this feature. (If there is neither possessor nor determiner then the grammar considers the **np-type**: if it is `COMMON`, it uses `NO` (for indefinite) and if it is `PROPER`, it uses `NOART`

  The number feature passed is the value passed from the determiner, if there is one, or the value from the current template.

  The person feature passed is the one from the current template.

Table 2. *Semantic rules of the conjunction, noun-phrase, and pronoun templates*

Template name	Semantic rules
CONJUNCTION	((this sentence) (this inh sentence)) ((this syn person) (CASE (UNION (first syn person) (second syn person)) OF ((first nil) (second nil) ((first second) second) ((first third) second) ((second third) second) (third third)))) ((this syn number) PLURAL) ((this syn gender) NEUTRAL) ((this syn definite) NOART) ((this syn sentence) (UNION (first syn sentence) (second syn sentence)))
NOUN-PHRASE	((this definite) (this inh definite)) ((this number) (this inh number)) ((determiner inh number) (this inh number)) ((this np-type) (this inh np-type)) ((this syn definite) (IF (AND (NULL (this possessor)) (NULL (this determiner))) THEN (UNION (this definite) (CASE (this np-type) OF ((common NO) (proper NOART)))) ELSE (UNION (this definite) (possessor syn definite) (determiner syn definite)))) ((this syn number) (UNION (determiner syn number) (this number))) ((this syn person) (this person)) ((this syn np-type) (CASE (this definite) OF ((NO COMMON) (NOART PROPER))))
PRONOUN	((this syn person) (this person)) ((this syn number) (this number)) ((this syn gender) (this gender)) ((this syn sentence) NO) ((this syn definite) NOART)

The np-type feature passed is `COMMON` if the value of definite is `NO` and `PROPER` if the value is `NOART`.

For the pronoun template, the grammar will:

- Pass up the **person**, **number**, and **gender** values from the current template (possibly using default values), along with the constraint that the string realized for it not be a sentence and not be preceded by an article.

Figure 7 shows a dependency graph for the conjunction template (a fragment of the larger input shown in figure 6). In this graph, an oval represents a template slot

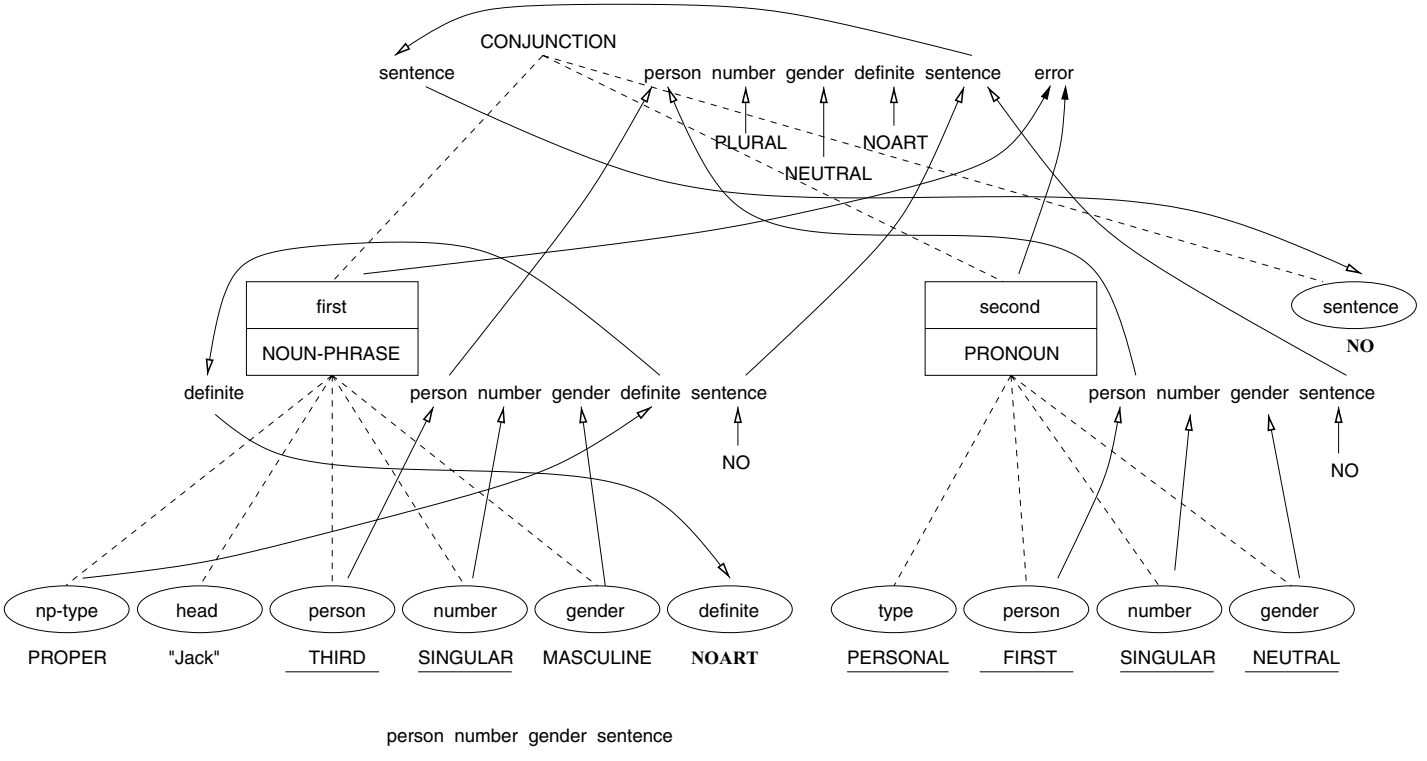

Fig. 7. Dependency graph of the conjunction template corresponding to the text *"Jack and I."*

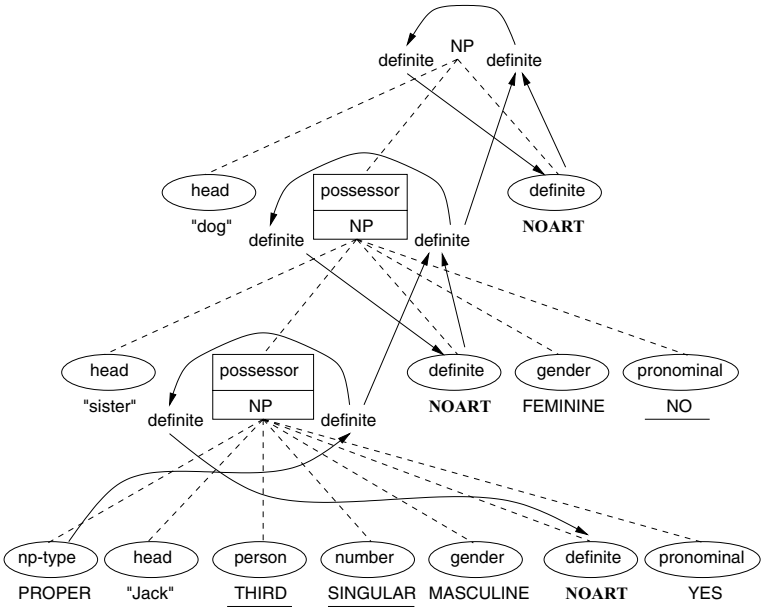

Fig. 8. Dependency graph of the noun-phrase template corresponding to the text *"his sister's dog."*

that is bound to an atomic value and a rectangle denotes a slot that is bound to another feature structure, where the top text specifies the slot name and and the bottom text is the name of a template assigned to this slot. A value with an underline means the value is the default for the slot. A value printed in bold font means that it is the rules of attribute evaluations. A dotted line specifies a hierarchical structure of a dependency graph constructed from a template. An arrow denotes the direction of the attribute propagation. For this input fragment, inherited attributes[7] have been initialized to the associated values given in an input. If the input does not specify a value for an inherited attribute, then the value `nil` is used.

The attribute evaluation is *depth-first*, and requires multiple traversals. Here, the noun-phrase sub-tree is evaluated twice, as we discover that the **definite** feature must be `NOART`. Since the pronoun template has no inherited attributes, a single evaluation would be sufficient. The conjunction sub-tree is also traversed twice because the **sentence** feature is re-assigned once (from `nil` to `NO`).

Figure 8 shows the tree and dependencies, for the fragment, *"his sister's dog"*. It shows how the definiteness of a noun phrase is dependent on the existence of a possessor. For example, if a possessor (such as *"his"* or *"Jack's"*) is specified, a noun phrase will not need an article. In this example, the **np-type** feature of the innermost noun phrase (*"Jack"*) has the value `PROPER`. This forces the **definite** feature to be `NOART`, according to the attribute grammar rule for the noun-phrase template.[8] The

[7] Inherited attributes are placed on the left side of each node. Synthesized attributes are on the right.

[8] The rule is shown in Table 2, as the fifth rule for the noun-phrase template.

```
((template noun-phrase)
 (head "book")
 (number PLURAL)
 (determiner ((template pronoun)
              (type DEMONSTRATIVE)
              (distance NEAR)
              (number SINGULAR)) )
)
```

Fig. 9. Feature structure with a constraint violation between the number of the head and the determiner.

value of the **definite** feature is then propagated up and forces the **definite** feature of outer noun-phrase template to be NOART.

Note that this feature structure can be generated differently as *"Jack's sister's dog"*, *"her dog"*, *"the dog of Jack's sister"*, *"the dog of his sister"*, and *"the dog of hers"*. While some of these variations require further investigation to determine how to transform a tree so that it reflects a new ordering of constituents, some can be implemented using semantic rules. For example, to avoid an awkward construction such as *"Jack's sister's dog"* in the sentence *"Jack and I want Jack's sister's dog to swim."*, in favor of *"his sister's dog"*, without the application having to request a pronoun explicitly, as in the example shown above, we could add a rule to force the **pronominal** feature of the inner most possessor to be YES, whenever a (repeated) noun phrase is a possessor of a possessor of the primary noun.

One important benefit of the use of attribute grammars is that they can help resolve inconsistencies in the input provided from an application. Previously, a generation system might not be able to recognize such conflicts, and therefore might generate a text that is ungrammatical, or it might simply fail to produce an output at all.

Figure 9 is an example input that has a conflict; the values of the **number** feature in the noun-phrase and pronoun templates are inconsistent. Executed literally, a generator would produce the phrase *"this books"*, rather than *"this book"* or *"these books"*. Figure 10 shows a dependency graph corresponding to the above input.

With the use of an appropriate attribute grammar (such as that shown in figure 2), an analysis of this structure would detect a conflict when the value SINGULAR of the **number** feature propagates upward and conflicts with the value PLURAL of the **number** feature of the noun-phrase template.

Once a conflict has been detected, a generator may choose to do one of three things:

1. It can stop the generation process and indicate the specific feature conflict that has occurred. For example, it can indicate that there is a conflict between the number feature of the determiner and between the number feature of the head noun in the noun-phrase template.
2. It can ignore the conflict, allowing the application inputs to override the grammar, and generate the ungrammatical output as specified. For example, it can produce the output "*this books*".

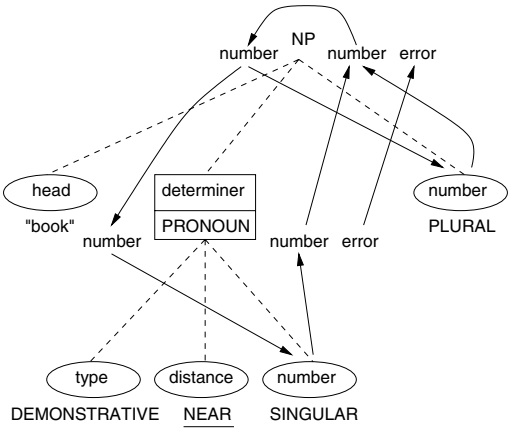

Fig. 10. Dependency graph corresponding to the Text *"this book"* or *"these books."*

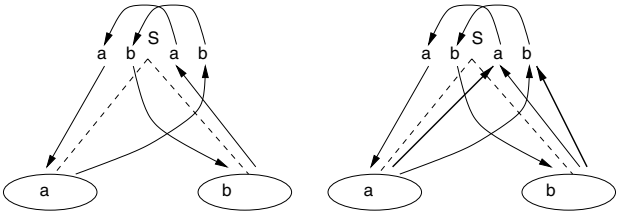

Fig. 11. Dependency graph that might cause an interchanging fixed-point.

3. It can attempt to fix the conflict to produce a grammatical output that is a plausible interpretation of the application designer's intent. For example, it could produce the output "*these books*".

There is a parameter within YAG to control which modes are to be used. The best modes to use depends on the use of the generator: during the development of application, it is useful to make errors apparent; by contrast, during the end-use of an application, it is most appropriate to fix errors silently (except for applications that need to generate un-grammaticalities for literary effect). The third choice (to correct an apparent error) can be implemented by applying a set of heuristic rules such as "Always favor the grammatical features of the head of a phrase over its specifier." and "Always favor the grammatical features of the subject of a clause over the features of the predicate."

Another class of conflicts are circular dependencies. This is a situation where the input fails to reach a state that does not change (a fixed-point) after successive propagation rules are applied. This will happen if there are some attributes whose semantic rules simultaneously alter the value of each attribute in a way that the values keep changing back and forth during attribute evaluation. In fact, a fixed-point has been reached but it is not a singleton (i.e. it is a set of two or more elements). This is called an *Interchanging Fixed-Point*. Figure 11(a) illustrates an example dependency graph where an interchanging fixed-point might be found.

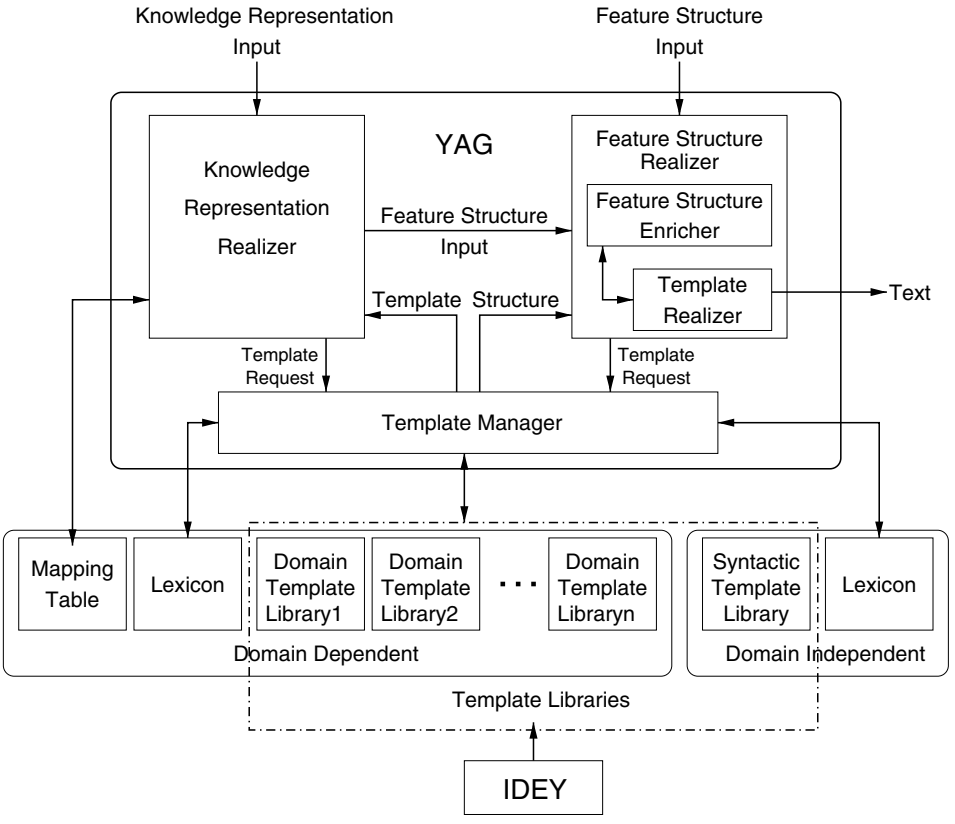

Fig. 12. YAG architecture.

This problem is solved by adding another dependency that implements a union of its current value and results from other dependencies, shown with a thick arrow in figure 11(b). However, it is a grammar writer's responsibility to identify the attributes and rules that might cause interchanging fixed-points and to add the additional dependencies where needed.

3 Implementation

Our implementation of YAG has a layered architecture as shown in figure 12. 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.

In this section, we discuss the implemented YAG architecture, a graphical development environment for YAG, a Java-based implementation of YAG, a a discussion of some practical experiences using YAG.

```
(member-class
 ((decl
    (be (template member-class)
        (slot-map ((class class)
                   (member member)) )
        (feature nil)
    )
 )))
```

Fig. 13. Simplified mapping entry of the MEMBER-CLASS case frame.

3.1 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 *Feature-Structure Enricher* mediates between the knowledge representation realizer and the feature structure realizer. It retrieves attribute grammars associated with each template to enrich missing information that is left unspecified by the knowledge representation realizer or directly from an application. However, this enrichment is optional.

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.

3.2 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*, *Domain Template Libraries*, and the *Domain Lexicons.* The domain independent knowledge base includes the *Syntactic Template Library* and the general *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. A sample entry of a mapping table is given in figure 13.

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 IDEY (The *I*ntegrated *D*evelopment *E*nvironment for *Y*AG) which is a program that allows a knowledge engineer to author and test templates in a graphical environment. (IDEY is described in section 3.3.)

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.

3.3 YAG development environment

We have implemented a graphical editing tool for YAG, called IDEY (**I**ntegrated **D**evelopment **E**nvironment for **Y**AG). (A developer can also modify existing templates or create new ones by editing the template definitions by hand. These definitions are stored as text files in both YAG's template language and in XML.) IDEY's graphical interface allows developers to edit templates through direct manipulation and to test newly constructed templates easily. The interface also helps prevent errors by constraining the way in which templates may be constructed or modified. For example, values of slots in templates are constrained by context-sensitive pop-up menu choices. A screen shot of IDEY is given in figure 14.

The interface of IDEY consists of two main areas, the project view and the template view, which we describe below.

Project View The top left section of the IDEY interface (figure 14) is the project view. It shows the resources currently included in a project. This includes template libraries, morphology functions, and lexicons. The bottom left section of the project view shows the contents of the currently selected resource.

Template View To the right of the project view is the template view. It is helpful when one creates a new template or selects an existing template for editing and testing. This view contains several tabbed panels used for specifying template components (such as template rules and template slots).

- The *Rule* tab gives the user access to template rules. A hierarchical (tree-based) interface helps developers visualize and navigate template rules. The rules shown on this panel can be modified by direct manipulation.
- The *View* tab shows the template rules shown by the rule tab in a declarative format (as they would appear in a text file).

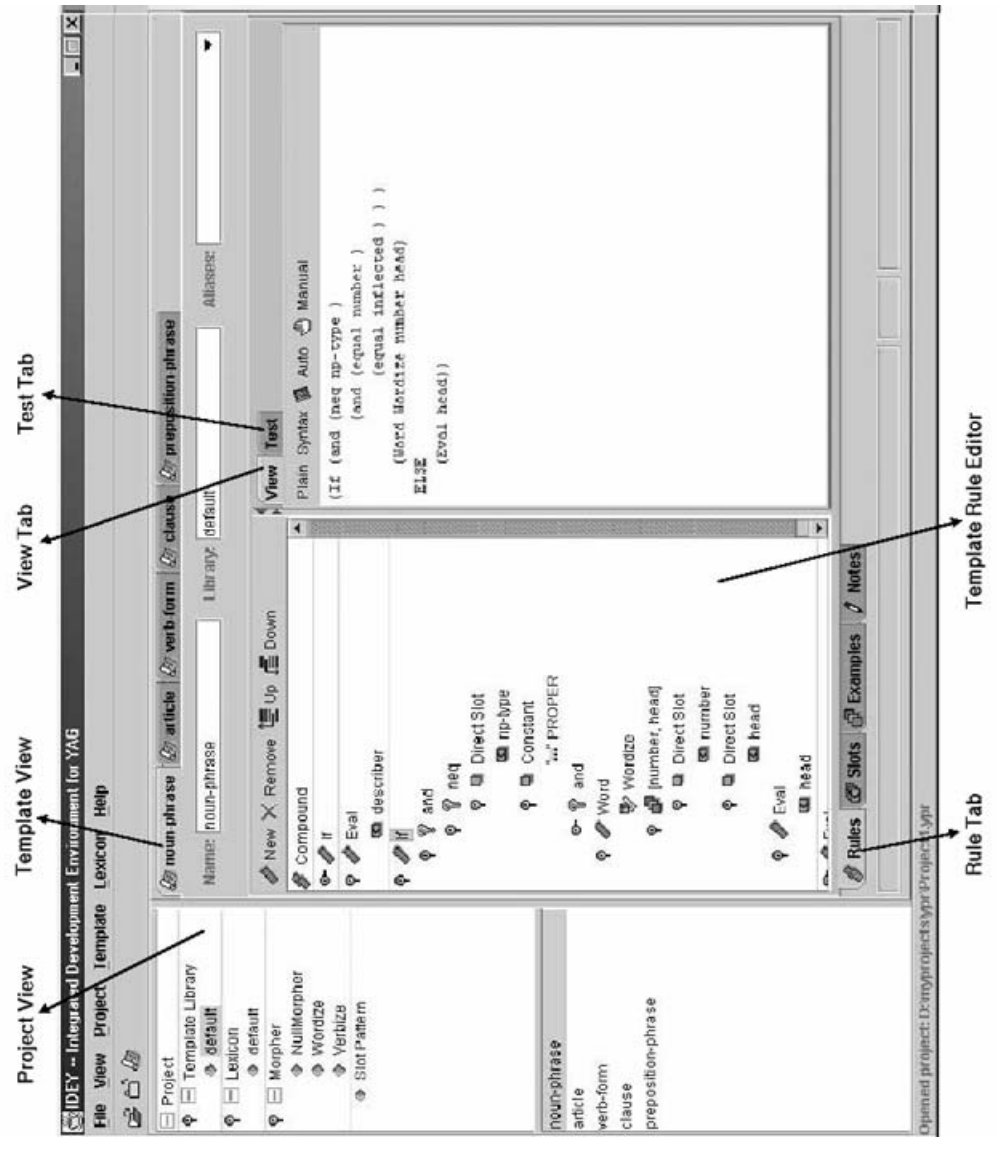

Fig. 14. A Screen shot of IDEY.

- The *Test* tab allows a user to enter values into a feature structure associated with the template being edited and then see what text would be realized.

3.4 The Java Implementation of YAG

There are two implementations of YAG, one in Lisp and one that uses Java. The Java implementation, called **JYAG** (Java 2.0 Platform **YAG**), provides an application program interface (API) that allows applications to realize text from feature structures.

Resources for JYAG, such as templates, lexicons, and morphology functions, can be created and saved into a single file that contains an instantiation of JYAG's container class called `Generator` using IDEY (the graphical development tool for YAG). (This approach makes use of Java's serialization technique.) Another way to create resources for JYAG is to express them in XML. The XML files can be created by hand, or using IDEY. IDEY can also be used to read in previously created XML resources for further authoring. Templates created for YAG (the lisp implementation) and templates created for JYAG can be used by either system by using IDEY to convert between the declarative format used by YAG and the XML format used by JYAG.

After the resources have been created, Java applications can access YAG by creating a `Generator` object and loading the contents from the saved file at run-time. To realize a text, applications create an input as a feature structure (an object of the `FeatureStructure` class), and pass it as an argument to a generator, using the generator function `realize`.

3.5 Practical experiences with YAG

Installing Lisp YAG or Java YAG (JYAG) is quick and easy. Installing the software takes less than a minute. Lisp YAG and JYAG can be easily integrated into an existing application. Both versions come with a pre-defined template set that covers basic English grammar. Application programmers can also define additional templates to suit their application's needs. For Lisp YAG, the input feature structure is realized by calling a function that returns a string as a result. For JYAG, a feature structure input can be constructed either using the JYAG API or from an XML input.

We have used YAG in building a system for simulating disagreements between two agents. This system, called IAS (Intelligent Argumentation System), uses YAG to generate the system's rebuttals to arguments made by the user. To use YAG from IAS, we first specified a mapping from the knowledge representation (a semantic network) into feature structures. We also had to author a small number of new templates. The speed to realize utterances was sufficient for real-time interaction.

The run-time of both versions of YAG is linear in the length of sentences to be generated. The realization speed depends on the number of templates and the depth of embedded feature structures used to generate a text. Table 3 includes a summary

Table 3. *Benchmark tests using YAG and JYAG*

Test ID	Embedded feature structures	Depth	Lisp YAG (seconds)	JYAG (seconds)
t1-1	0	0	0.016	0.006
t3-0	2	1	0.072	0.014
t2-2	2	1	0.072	0.015
t2-3	3	1	0.087	0.018
t3-1	4	2	0.141	0.026
t2-5	5	3	0.155	0.031
t2-6	6	3	0.128	0.034
t3-2	6	3	0.220	0.043
t2-7	7	3	0.156	0.039
t3-3	8	4	0.310	0.057

of some bench-mark tests that we performed to verify YAG's linearity.[9] For these tests, we used a machine with a Pentium IV 1.5 GHz Processor, 512 MB RAM, and Windows 2000 Professional. For even the hardest cases, YAG generated sentences in less than half a second, and JYAG typically takes about about one fifth the time that JYAG does.

We have distributed Lisp YAG to a few other researchers. They found that while it was fairly easy to install and use YAG from an application, they had difficulty defining new templates. These experiences led us to design and implement the graphical development environment for YAG, discussed above.

4 Realizing an input with YAG

This section presents the text realization algorithms of YAG from end-to-end, following the realization of a single example, *"John's blood pressure is uncontrolled.* We will begin with an input constructed in a knowledge representation language, so that we may consider the realization process from the outermost layer of YAG (the Knowledge Representation Realizer). We consider the choices that the application must make in constructing this input. The input will then be mapped onto a (partially specified) feature structure, using information from the mapping table. Then, YAG's feature-structure enricher will be used to enrich the top-level feature structure. Finally, we see the final realization that is produced by YAG's Feature Structure Realizer, including an additional call to the feature-structure enricher to process a feature structure within an embedded template.

4.1 Constructing the knowledge representation input

Figure 15 shows an example SNePS network (see section 2.2.2) that expresses the content *"John's blood pressure is uncontrolled."*.[10] This network includes several case

[9] The inputs for these test cases are given in on our website, `tigger.cs.uwm.edu/nlkrrg/testcases`.

[10] In this figure, the "!" character attached to the top-level node (`M4`) means that this proposition is believed by the SNePS agent. The use of the `LEX` arc is a mechanism used to

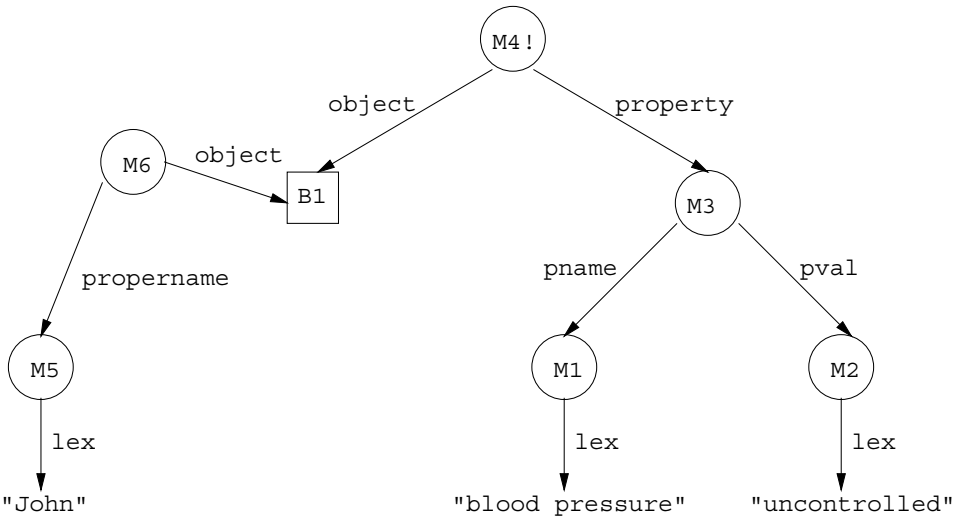

Fig. 15. Semantic network of the proposition *"John's blood pressure is uncontrolled."*

frames (OBJECT-PROPERTY, OBJECT-PROPERNAME, PNAME-PVAL). Since a semantic network represents a set of propositions, in no particular order, it is important for an application to identify the main proposition to be realized and the type of syntactic unit that is desired, because this network can be used to generate different phrases or sentences depending on what selections are made. For example, the semantic network of figure 15 can be expressed as a declarative sentence *"John's blood pressure is uncontrolled."* or *"John has uncontrolled blood pressure."*, or as a question *"Is John's blood pressure uncontrolled?"*, or a noun phrase *"John's uncontrolled blood pressure"*.

In this example, the main proposition is M4. To capture this, we express the network as the following ordered list of propositions:

```
((M4! (OBJECT B1)
      (PROPERTY (M3 (PNAME (M1 (LEX "blood pressure")))
                    (PVAL (M2 (LEX "uncontrolled"))))))
 (M6 (OBJECT B1) (PROPERNAME (M5 (LEX "John")))) )
```

Next, we must add control features to indicate that the output should be a complete, declarative copulative sentence. The appropriate control features for this are `form = decl` and `attitude = be`.

The knowledge representation input for this example is thus the following:

```
(((M4! (OBJECT B1)
       (PROPERTY (M3 (PNAME (M1 (LEX "blood pressure")))
                     (PVAL (M2 (LEX "uncontrolled"))))))
  (M6 (OBJECT B1) (PROPERNAME (M5 (LEX "John")))) )
 ((form decl)
  (attitude be) ) )
```

distinguish a concept node from the string that names it. These are common occurrences in SNePS networks that were omitted from the simplified examples shown earlier.

This input does not provide all the necessary linguistic information (e.g. *person*, *number*, and *gender*) for realization, however in later steps, enriching the partially specified feature structure and instantiating a template, default values will be added. If an application wanted to override a default, such as to pronominalize some discourse entity, it could add additional control features to the knowledge representation input.

4.2 Mapping the knowledge representation input onto a feature structure

During the mapping process, each proposition in the input structure is used to select appropriate templates for realization. As mentioned above, the first proposition in the list is taken as the primary proposition (others will be used to modify or specify nodes in the primary proposition). The primary proposition, along with the control features are used as an index into a mapping table that selects a template for realization and a mapping between the arguments to the proposition and the feature-value pairs in the feature structure. In our example the primary proposition is an OBJECT-PROPERTY proposition. Below is the fragment of the mapping table that specifies the template that realizer should use to express `OBJECT-PROPERTY` propositions:

```
(object-property
 ((decl (be (template object-property)
            (slot-map ((object agent)
                       (property property)) )
            (feature nil)
        ) )
  (int  (be (template object-property)
            (slot-map ((object agent)
                       (property property)) )
            (feature ((mood yes-no)) )
        ) )
 ) )
```

This fragment contains two choices for `OBJECT-PROPERTY` propositions; which one to use is determined by the control features. In our example, the control features specify `(form decl)` and `(attitude be)`, so the first entry is the one that is selected. The Knowledge Representation Realizer uses the **template** feature of the matched entry to map that the current proposition to the object-property template.[11] The **slot-map** feature of an entry in the Mapping Table provides information on how to map each arc of the proposition to slots in the template. Here, `((object agent) (property property))` tells the Knowledge Representation Realizer to assign the value of the `object` arc to the **agent** slot, and the `property` arc to the **property** slot.

Based on the information in the Mapping Table, the following feature structure is constructed:

[11] It is not required that the names of a case frame and its associated template be the same.

```
((template object-property)
 (agent B1)
 (property (M3 (PNAME (M1 (LEX "blood pressure")))
               (PVAL (M2 (LEX "uncontrolled")))) )
)
```

Now, the Knowledge Representation Realizer must realize the value assigned to each constructed slot. It does this by recursively repeating the process of using the mapping table to map each case frame in the input to some template, for each of the remaining case-frames in the input. If there is no mapping table for a case frame, then YAG attempts to process each node separately, replacing the feature that cannot be mapped directly. For our example, the `property` feature has a value that includes an embedded case frame, but this case frame (`PNAME-PVAL` has no entry in the mapping table. The arcs `PNAME` and `PVAL` are mapped onto the features `pname` and `pval`, respectively. The embedded nodes contain the `LEX` case frame, which is always mapped onto whatever is the value at the end of the `LEX` arc. So, in our example, `"blood pressure"` replaces `(M1 (LEX "blood pressure"‘))`, and `"uncontrolled"` replaces `(M2 (LEX "uncontrolled"))`. The revised feature structure is the following:

```
((template object-property)
 (agent B1)
 (pname "blood pressure")
 (pval "uncontrolled")
)
```

The next step is to replace discourse entities from the primary proposition with a description from the remaining propositions. The value of the `agent` slot, `B1`, is the base node corresponding to the discourse entity *"John"*. The Knowledge Representation Realizer needs to replace it with an appropriate feature structure. It does that by searching the supporting propositions (the rest of propositions in the proposition list) that contain information about `B1`. In this case, the related proposition is:

```
(M6! (OBJECT B1) (PROPERNAME (M5 (LEX "John"))))
```

The `OBJECT-PROPERNAME` case frame is one of the pre-defined case frames that, if it is not the primary proposition, the pre-defined feature structure will be used. Here, the `OBJECT-PROPERNAME` case frame is treated as a proper noun. The feature structure that replaces `B1` is as follows:

```
((template noun-phrase)
 (np-type proper)
 (head "John") )
```

Furthermore, the Knowledge Representation Realizer must check the feature list in the input to see if there is a feature to add or if it needs to override the features of this base node. Since there are no such features in this example, the feature structure remains unchanged. Figure 16 shows the feature structure that is finally used as an input for the Feature Structure Realizer to realize the text *"John's blood pressure is uncontrolled."*.

```
((template object-property)
 (agent ((template noun-phrase)
         (np-type proper)
         (head "John")) )
 (pname "blood pressure")
 (pval "uncontrolled") )
```

Fig. 16. The (partially specified) feature structure for realizing *"John's blood pressure is uncontrolled."*

4.3 Enriching a partially specified input

As mentioned in section 2.3, feature structures constructed from a knowledge representation are often under-specified, because very few syntactic constraints exist at the knowledge representation level. The text realizer must provide reasonable values for the missing information.

In addition to a typical defaulting mechanism similar to what is used in most text realization systems, YAG's also includes a mechanism for enriching partially specified inputs before passing them to the Feature Structure Realizer. The feature-structure enricher makes use of attribute grammar rules associated with each template. Using the example input shown in figure 16, YAG's enricher constructs a tree data structure that is equivalent to the given input. Attribute grammar rules are then retrieved for the top-level template (object-property) and any embedded feature structures visible from the top-level (noun-phrase). Since there is no rules specified for the object-property template, only the rules for the noun-phrase template (given in Table 2) are evaluated.

The evaluation of attributes for the noun-phrase template reveals that the **definite** slot should be bound to `NOART`. This binding is specified by the following semantic rule, which deals with the definiteness feature of a noun phrase:

```
((this syn definite) (IF (AND (NULL (this possessor))
                              (NULL (this determiner))) THEN
                         (UNION (this definite)
                                (CASE (this np-type) OF
                                      ((common NO)
                                       (proper NOART)))  )
                    ELSE
                         (UNION (this definite)
                                (possessor syn definite)
                                (determiner syn definite)) ))
```

As the **possessor** and **determiner** slots are not specified, in our example, the **definite** synthesized attribute (`(this syn definite)`) is bound to the union of the current **definite** slot (which is `nil`) and the value `NOART`. (The value of **np-type** slot has been bound to `PROPER`.) After the second iteration of attribute evaluations, no changes are discovered, so the evaluation algorithm terminates returning the enriched feature structure shown in figure 17 for realization.

```
((template object-property)
 (agent ((template noun-phrase)
         (np-type proper)
         (definite NOART)
         (head "John")) )
 (pname "blood pressure")
 (pval "uncontrolled") )
```

Fig. 17. Complete feature structure for realizing *"John's blood pressure is uncontrolled."*

4.4 Realizing the enriched feature structure

Realizing a feature structure involves three steps. First the template named in the input is instantiated to assign values to each of the template slots. Then the realizer considers the value assigned to each template slot, if a value is a feature structure, then the feature structure is recursively realized to obtain a string. Finally, the top level template is realized by interpreting each of the template rules within the instantiated template.

4.4.1 Instantiating a template

Feature structures are instantiated by the Template Manager. The Template Manager searches for the **template** feature in the feature structure of figure 17 to find the appropriate template to use. Since, in our example, the **template** feature is bound to `object-property`, the Template Manager retrieves the object-property template from the template library. The structure of this template is shown in figure 18. (Recall that the ^ symbol is used to distinguish symbolic values from the names of template slots, as in the `TEMPLATE` rule in the figure.)

The value of each feature in the input (figure 17) is assigned to the corresponding template slot.[12] Here, the Feature Structure Realizer assigns the value of `agent` in the input to the **agent** slot of the template, `pval` to the **pval** slot, and `pname` to the **pname** slot. The **property** and **sentence** slots remain unchanged since there is no assignment for them. Thus, the defaults from the template definition are applied automatically to the slots that are not specified in the input. The instantiated template slots are shown in figure 19.

4.4.2 Realizing slot values

Since YAG permits recursive templates, a feature structure can be bound to any of the template slots. The Feature Structure Realizer has to realize those feature structures to surface strings prior to the realization of the current feature structure.

In our example, the value of the **agent** slot is following feature structure:

```
((template noun-phrase)
 (np-type proper)
 (definite NOART)
 (head "John") )
```

[12] The slot must either have the same name as the feature or have the feature as one of its specified aliases.

```
(def-template
  :name 'object-property
  :alias nil
  :library 'semantic
  :slot '((name agent)
           (default-value nil))
         ((name pval)
          (default-value nil))
         ((name pname)
          (default-value nil))
         ((name property)
          (default-value nil))
         ((name sentence)
          (default-value yes))
        )
  :rule '((IF (equal pname nil)
              (TEMPLATE clause
                        ((mode DECLARATIVE)
                         (process-type ASCRIPTIVE)
                         (carrier ^agent)
                         (attribute ^property)
                         (sentence ^sentence)) )
           ELSE
              (TEMPLATE clause
                        ((mode DECLARATIVE)
                         (process-type ASCRIPTIVE)
                         (carrier ((template noun-phrase)
                                   (head ^pname)
                                   (possessor ^agent)))
                         (attribute ^pval)
                         (sentence ^sentence)) )
         ) )
  :sample '("John is good." "John's age is 20.")
)
```

Fig. 18. Object-Property template.

```
((agent ((template noun-phrase)
         (np-type proper)
         (definite NOART)
         (head "John")) )
 (pval "uncontrolled")
 (pname "blood pressure")
 (property nil)
 (sentence yes) )
```

Fig. 19. Instantiated template slots for realizing *"John's blood pressure is uncontrolled."*

It will be realized using the same process for template instantiation. First, the Template Manager retrieves the noun-phrase template (figure 20) from the template library.

The noun-phrase template is instantiated. Its slots are shown below:

```
((head "John")          (countable YES)
 (np-type proper)       (person THIRD)
 (determiner nil)       (number SINGULAR)
 (describer nil)        (gender MASCULINE)
 (qualifier nil)        (inflected YES)
 (definite NOART)       (possessor nil)
 (regular-noun YES))
```

```
(def-template
  :name 'noun-phrase
  :alias nil
  :library 'grammar
  :slot '(((name head)
           (default-value nil) )
          ((name np-type)
           (default-value COMMON) )
          ((name determiner)
           (default-value nil) )
          ((name describer)
           (default-value nil) )
          ((name qualifier)
           (default-value nil) )
          ((name definite)
           (default-value NO) )
          ((name regular-noun)
           (default-value YES) )
          ((name countable)
           (default-value YES) )
          ((name person)
           (default-value THIRD) )
          ((name number)
           (default-value SINGULAR) )
          ((name gender)
           (default-value NEUTRAL) )
          ((name inflected)
           (default-value YES) )
          ((name possessor)
           (default-value nil) ))
  :rule
   '((IF (and (not (equal definite NOART))
              (not (equal countable NO)))
         (TEMPLATE article
                   ((definite ^definite)
                    (regular-noun ^regular-noun)
                    (number ^number)
                    (countable ^countable)) ))
     (IF (not (equal possessor nil))
         (TEMPLATE possessor
                   ((possessor ^possessor)
                    (pronominal (possessor pronominal))) )
      ELSE
         (EVAL determiner) )
     (EVAL describer)
     (IF (and (not (equal np-type PROPER))
              (equal inflected YES))
         (LEX WORDIZE number head)
      ELSE
         (EVAL head) )
     (EVAL qualifier)
    )
  :sample '("a dog" "John" "the dog that I kicked")
)
```

Fig. 20. Noun-phrase template.

The Template Realizer then processes each rule in the instantiated noun-phrase template. The first `IF` rule produces no output because its condition is not met. The second `IF` rule produces no output because its condition is not met and the value with the `ELSE` part is `nil`, because no determiner has been specified. Similarly, because no describer was specified, the third rule, `(EVAL describer)`, also produces no output. The fourth rule, another `IF` rule, produces the string *"John"* which is the result of evaluating the *ELSE* part, `(EVAL head)`. The last rule, `(EVAL qualifier)`, returns no result. Finally, this feature structure yields the surface string *"John"*. This

string is inserted in the **agent** slot. The template slots after this realization are shown below:

```
((agent "John" ((template noun-phrase)
                (np-type proper)
                (definite NOART)
                (head "John")) )
 (pval "uncontrolled")
 (pname "blood pressure")
 (property nil) )
```

We keep the realized feature structure as well as its surface string because template rules from the top-level template (or other embedded templates) can refer to the slots of this feature structure.

4.4.3 Realizing the top-level template

After all embedded feature structures have been realized, then the Template Realizer can interpret the template rules in the top-level template. (In our example, this is the object-property template. The results from the realization of each rule are then combined, yielding the final surface string.

The template rules are interpreted in the order in which they appear: The `IF` rule checks its condition `(EQUAL pname nil)`, and it fails. Therefore the second rule is selected. It is shown below:

```
(TEMPLATE clause
          ((mode DECLARATIVE)
           (process-type ASCRIPTIVE)
           (carrier ((template noun-phrase)
                     (head ^pname)
                     (possessor ^agent)) )
           (attribute ^pval)
           (sentence ^sentence)) )
```

This rule constructs a feature structure for the clause template with the specified parameters. The constructed feature structure is:

```
((template clause)
 (mode DECLARATIVE)
 (process-type ASCRIPTIVE)
 (carrier ((template noun-phrase)
           (head "blood pressure")
           (possessor "John" ((template noun-phrase)
                              (np-type PROPER)
                              (definite NOART)
                              (head "John")))) )
 (attribute "uncontrolled")
 (sentence YES) )
```

Before final realization, the system evaluates the attribute grammar rules for this structure, which results in the feature-value pair `(definite NOART)` being added into the embedded feature structure that is bound to the **carrier** slot. (This feature is

specified by the same semantic rule given earlier (in section 4.3), which controls the dependency between definiteness and the occurrence of a possessor or determiner.)

The realization of the enriched feature structure starts by retrieving and instantiating the clause template. The **carrier** slot is the only slot that needs realization since its value is a feature structure.

The following feature structure is realized with the noun-phrase template. Its surface string, *"John's blood pressure"*, is used to fill the **carrier** slot:

```
((template noun-phrase)
 (head "blood pressure")
 (definite NOART)
 (possessor "John" ((template noun-phrase)
                    (np-type PROPER)
                    (definite NOART)
                    (head "John"))) )
```

The revised feature structure is shown below:

```
((template clause)
 (mode DECLARATIVE)
 (process-type ASCRIPTIVE)
 (carrier "John's blood pressure"
          ((template noun-phrase)
           (head "blood pressure")
           (definite NOART)
           (possessor "John" ((template noun-phrase)
                              (np-type PROPER)
                              (definite NOART)
                              (head "John")))) )
 (attribute "uncontrolled")
 (sentence YES) )
```

The Template Realizer realizes the clause template with the above feature structure and finally returns *"John's blood pressure is uncontrolled."* as the surface string.

5 Related work

Previous papers by the authors have discussed the use of attribute grammars to address underspecified inputs, surveyed the status of current work in text realization, and overviewed the architecture of YAG (Channarukul *et al.* 2000; McRoy and Channarukul 2001; Channarukul, McRoy and Ali 2001). Here we have described the design and use of YAG in detail, including the interactions between YAG and an implemented attribute-grammar structure enricher. We also introduced some new tools, IDEY (a graphical development environment for YAG) and JYAG (a Java-based implementation of YAG), and provide some practical results based on the use of YAG.

The most similar work to ours is the work by Busemann and Horacek on TG/2 (Busemann 1996; Busemann and Horacek 1998). TG/2 employs a rule-based technique to control the realization of templates. Both YAG's templates and TG/2's rules allow specifications of canned texts, templates, and syntactic representations.

TG/2's templates can require backtracking, however, because multiple rules may apply. Careful rule definition in TG/2 can avoid backtracking; YAG never requires 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), while YAG implements a more powerful template-based approach through a number of extensions (see section 2.1). YAG's template language is also more declarative, yielding higher maintainability and comprehensibility.

Other recent template-based systems include D2S (Theune, Klabbers, Odijk, de Pijper and Krahmer 2001), ECRAN (Geldof and de Velde 1997), and MacroNode (Not and Zancanaro 2000). In these systems, the general approach to templates is restricted to traditional methods, such as slot-filling and hence cannot support the variations addressed by YAG. These systems also address a different problem than YAG, namely the planning of multi-sentential texts prior to realization. As such, their templates address issues important to text planning (such as how to decide what to say and how to create a sense of coherence), rather than low-level syntactic constraints (such as number agreement or choosing the right pronoun). At the point of realization, these systems output a concatenation of canned texts and text-fragments that have been generated by procedures separate from the template mechanism. These approaches are much more domain-dependent than YAG. However, they do demonstrate that templates can be used for multi-sentence texts, which we have not yet tried to do with our system.

Other older template-based systems include TEXT (McKeown 1982, 1985a, 1985b), which uses templates primarily for the generation of a complete discourse (multiple sentences or a paragraph), and purely syntactic grammar-based systems such as MUMBLE (Meteer, McDonald, Anderson, Forster, Gay, Huettner and Sibun 1987) and RealPro (Lavoie and Rambow 1997).

The primary alternative to template based approaches are systems that rely on unification. One well-known and highly regarded system is FUF/SURGE (Elhadad 1992, 1993; Elhadad and Robin 1996), which uses a functional unification grammar to realize text. In these systems, the speed of realization is dependent on the size of the grammar, rather than the intended output. As a result, template-based realization systems such as YAG better meet the needs of real-time systems because they do not require extensive search through a large grammar. YAG is appropriate when generation speed is crucial and fine-grained control of text is not necessary (or desirable), because YAG is able to make extensive use of defaults and other pre-specified information. (It can also provide fine-grained control, when required.)

6 Conclusion

In this paper, we have presented a *template-based approach* to text realization that meets the requirements of real-time, interactive systems such as a dialog system or an intelligent tutoring system. Such applications require that a realizer be fast, robust, and expressive. Our augmented template-based approach can generate text at high speed, when compared with other sophisticated approaches.

This paper also addresses issues of text quality, generality and flexibility that have been the key limitations of previous template-based approaches. Since traditional template-based approaches rely solely on slot-filling mechanism, the quality of generated text is limited by a grammar (if one exists) that is hard-coded procedurally in a text generator. This grammar is then incomprehensible and hard to modify or extend. The augmented template-based approach that is presented herein suggests modifications to the traditional approach to allow a declarative representation of templates that does more than simple slot-filling. Template rules that provide more control over the generation of text are introduced. They cover most tasks that would be required to generate texts in general. The tasks range from string manipulation rules, recursive calls to other templates, to conditional expressions within a template to determine how a text should be generated based on the values defined in template slots.

Such modifications result in a better approach that allows various text structures or templates to be defined with the same formalism. It allows declarative representations of grammar that do not depend upon the code that executes them, yielding a grammar that is more comprehensible and easy to extend and modify. Moreover, templates can also be designed to bridge the gap between a semantic representation of an application and the structure of corresponding text that is more syntactic, so that an application will have less work to do in providing syntactic information to the text generator.

Handling partially-specified input has also been another important issue that contributes to the robustness of a generator. Our approach uses both defaulting mechanism and attribute grammars with semantic propagation rules. They allow a generator to improve the quality of input to the generator for better realization.

Lastly, we have presented YAG (Yet Another Generator), a text generator that implements our augmented template-based approach. It allows an application to generate text from either a knowledge representation or a feature structure that is independent of any knowledge representation.

Acknowledgements

The authors thank the anonymous reviewers for their many helpful comments. We also acknowledge the financial support of the National Science Foundation (under grants IRI-9701617 and DUE-9952703), Wright State University (for NSF BCS-9980054) and Intel Corporation.

References

Alblas, H. (1991) Introduction to attribute grammars. In: Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems: Lecture Notes in Computer Science 545*, pp. 1–15. Springer-Verlag.

Busemann, S. (1996) Best-first surface realization. In: Scott, D., editor, *Proceedings Eighth International Workshop on Natural Language Generation*, pp. 101–110.

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

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

Channarukul, S., McRoy, S. and Ali, S. (2001) YAG: A template-based text realization system for dialog. *J. Uncertainty, Fuzziness, and Knowledge-Based Syst.* **9**(6): 649–659.

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*, pp. 163–170. Israel.

Elhadad, M. (1992) *Using argumentation to control lexical choice: A functional unification-based approach.* PhD thesis, Computer Science Department, Columbia University.

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

Elhadad, M. and Robin, J. (1996) An overview of SURGE: a reusable comprehensive syntactic realization component. Technical Report 96-03, Department of Computer Science, Ben Gurion University, Beer Sheva, Israel.

Geldof, S. and de Velde, W. V. (1997) An architecture for template based (hyper)text generation. In: Brusilovsky, P., Stock, O. and Strapparava, C., editors, *Proceedings 6th European Workshop on Natural Language Generation (EWNLG'97)*, pp. 28–37. Duisburh, Germany.

Grosz, B. J., Sparck-Jones, K. and Webber, B. L. (1986) *Readings in Natural Language Processing*. Morgan Kaufmann.

Halliday, M. (1994) *An Introduction to Function Grammar*. Edward Arnold.

Hovy, E. (1997) Language generation: overview. In: *Survey of The State of the Art in Human Language Technology*, pp. 139–146. Cambridge University Press.

Kay, M. (1979) Functional grammar. *Proceedings 5th Annual Meeting of the Berkeley Linguistic Society*, pp. 142–158. Berkeley, CA.

Knight, K. and Hatzivassiloglou, V. (1995) Two-level, many-paths generation. *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pp. 252–260. Cambridge, MA.

Knuth, D. E. (1968) Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2): 127–145. (Correction: *Mathematical Systems Theory*, **5**(5): 95–96 (March 1971).)

Lavoie, B. and Rambow, O. (1997) A fast and portable realizer for text generation systems. *Proceedings Fifth Conference on Applied Natural Language Processing*, pp. 265–268. Washington, DC.

Levison, M. and Lessard, G. (1990) Application of attribute grammars to natural language sentence generation. In: Deransart, P. and Jourdan, M., editors, *Attribute Grammars and their Applications (WAGA): Lecture Notes in Computer Science 461*, pp. 298–312. Springer-Verlag.

Mann, W. C. (1983) An overview of the Penman text generation system. *Proceedings Third National Conference on Artificial Intelligence (AAAI-83)*, pp. 261–265. Washington, DC. (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. *Proceedings 20th Annual Meeting of the ACL*, pp. 113–120. Toronto, Ontario, Canada.

McKeown, K. R. (1985a) Discourse strategies for generating natural-language text. *Artif. Intell.* **27**(1): 1–42.

McKeown, K. R. (1985b) *Text Generation: Using discourse strategies and focus constraints to generate natural language text*. Studies in Natural Language Processing. Cambridge University Press.

McRoy, S. W. and Channarukul, S. (2001) Creating natural language output for real-time applications. *intelligence: New Visions of AI in Practice*, **12**(2): 20–34.

Meteer, M. W. (1990) *The "Generation Gap" The Problem of Expressibility in Text Planning*. PhD thesis, University of Massachusetts.

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.

Not, E. and Zancanaro, M. (2000) The macronode approach: Mediating between adaptive and dynamic hypermedia. In *Proceedings of the International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, pp. 167–178. Trento, Italy.

Reiter, E. (1995) NLG vs. Templates. *Proceedings Fifth European Workshop on Natural Language Generation (ENLGW-95)*, Leiden, The Netherlands.

Reiter, E. and Dale, R. (1997) Building applied natural language generation systems. *Natural Lang. Eng.* **3**(1): 57–88.

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

Shapiro, S. C. and Rapaport, W. J. (1992) The SNePS family. *Comput. & Math. with Applic.* **23**(2–5).

Smith, G. W. (1991) *Computers and Human Language*. Oxford University Press.

Theune, M., Klabbers, E., Odijk, J., de Pijper, J. R. and Krahmer, E. (2001) From data to speech: a general approach. *Natural Lang. Eng.* **7**(1): 47–86.