SAG: A Procedural Tactical Generator for Dialog Systems

Dalina Kallulli

SAIL LABS Technology, Operngasse 20B, A-1040 Vienna, Austria
dalina.kallulli@sail-technology.com

Abstract

Widely used declarative approaches to generation in which generation speed is a function of grammar size are not optimal for real-time dialog systems. We argue that a procedural system like the one we present is potentially more efficient for time-critical real-world generation applications as it provides fine-grained control of each processing step on the way from input to output representations. In this way the procedural behaviour of the generator can be tailored to the task at hand. During the generation process, the realizer generates flat deep structures from semantic-pragmatic expressions, then syntactic deep structures from the deep semantic-pragmatic structures and from these syntactic deep structures surface strings. Nine different generation levels can be distinguished and are described in the paper.

1. Introduction

A major challenge for spoken dialog systems is the design of the system's natural language generation (NLG) module. This challenge arises from the fact that the generator needs to be sensitive and adjustable to many features of the dialog domain, user groups and dialog context [1]. Designers of NLG systems must normally consider the trade-off between the quality of text and the speed of realization. Turning to the unique needs of dialog systems, the generation speed is a central issue since interaction must occur in real-time. As has been pointed out in the literature [2], natural language realization systems can address this constraint in two ways: either the system designer must anticipate and specify all possible natural language outputs before runtime and supply the necessary program logic to produce the correct output at the correct time and hope that problems will never arise, or the system must be able to dynamically generate natural language outputs in real time.

There are a number of reasons why dynamic generation is to be preferred over simpler methods of text realization such as canned text and/or templates, on which the generation component of most dialog systems is based. While canned text systems are trivial to create, they are by their very nature inflexible (leading to user frustration and ultimately rejection) and wasteful on resources. Similarly, output produced by template-based generators lacks the variability and robustness so crucially needed by conversational systems. The ability to provide customized responses to users as well as issues hinging on the desirability of software reusability across applications are then obvious advantages of dynamic generation.

In the context of dialog systems, existing generators are either too slow (e.g. Penman [3], FUF/SURGE [4], [5]) since their approaches traverse the entire generation grammar rather than the input to be generated; or their grammar is too limited (e.g. TEXT [6]) leading to customization and portability problems; or realization is implemented as a production system (e.g. TG/2 [7], [8]) which is not suitable for real-time applications because of the inherently inefficient derivation of results in such systems; or realization is handled by employing statistical approaches which may provide ungrammatical results (e.g. Nitrogen [9], [10], [11]); or the realization system implements a template-based approach (e.g. YAG [1]) which again compromises quality of output.

In this paper we describe SAG (Sail Labs Answer Generator). SAG is a real-time, multilingual, general-purpose generation system that enables the use of dynamically generated natural language for dialog applications.[1]

Most existing approaches to realization are declarative. Declarative here means that the generation system defines a generic algorithm that controls how the input and grammar specifications are combined to yield the output of the generation process which can then be linearized into text. SAG differs from existing generators in that it employs a procedural approach. In contrast to declarative approaches, which define a set of conditions that have to hold in the outputs and rely on a generic algorithm to verify whether a certain input matches to a structure that fulfills these conditions, a procedural approach allows for the specification of a sequence of procedures from a set that should be applied to transfer inputs into outputs. The procedural approach provides fine-grained control of each processing step on the way from input to output representations. We argue that this is potentially more efficient for specific generation applications as the procedural behaviour of the generator can be tailored to the task at hand and that therefore a procedural system like SAG is better suited for time-critical real-world applications.

2. SAG Component Overview

In SAG, the tactical generation starts from semantic-pragmatic representations, which consist of a set of statements (predicates and meta-predicates) containing semantic, syntactic, pragmatic and morphological information that constitutes the content of the text to be generated. The elements of the representations are expressions in the so-called Simple Semantic-pragmatic Representation Language (SSRL). SSRL describes event and entity nodes of a network, their properties, and links among them [12], [13]. An example of the SSRL sequence that would ultimately generate the sentence "Do you want me to read the menu?" is provided in Figure 1. Figure 2 illustrates the resulting syntactic structure after all Lingware procedures (see section 3) have been executed. The output text is given in Figure 3.

[1] Though SAG was primarily designed for use within the Sail Labs Conversational System, it has also been employed for other real-time applications such as natural language interfaces to databases.

```
(state "x1" "want")
(experiencer "x2" "x1" "x3")
(hearer "x14" "x3" "you")
(action "x5" "read")
(agent "x8" "x5" "x6")
(entity "x6" "i")
(object "x7" "x5" "x16")
(entity "x16" "menu")
(tense "x1" pr)
(number "x16" sg)
(defness "x16" def)
(sentence-type "x1" main)
(sentence-type "x5" infin)
(smood "x1" qyn)
```

Figure 1: SSRL sequence

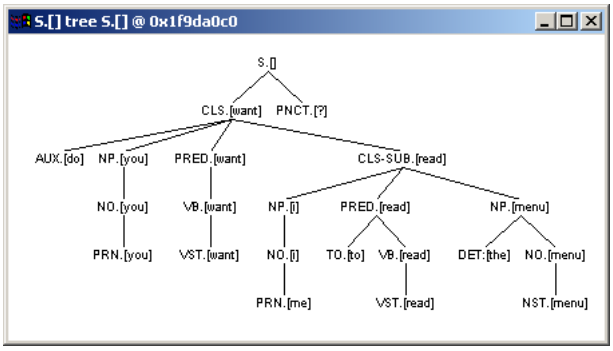

Figure 2: Resulting syntactic structure

```
"Do you want me to read the menu?"
```

Figure 3: Text output

The SSRL statements are created by a text-planning process, the strategic component, which generates the so-called "WHAT" [6]. SSRL statements serve as an interface between strategic and tactical generation.

The tactical generator uses a programming environment developed at Sail Labs, which uses a Lingware intepreter written, for efficiency reasons, in C++. Lingware is the programming language used to write the generation grammar. It provides a set of LISP and built-on-LISP functions, which represent linguistic operators for natural language processing. In what follows, program parts of the generation grammar are called (Lingware) procedures.

Each expression of the representation language has a corresponding Lingware procedure with the same name. Thus, the names of the SSRL expressions simultaneously stand for representation language units and program parts of the tactical generator.

The sequences of SSRL statements are prompted by actions specified by the dialog manager. Specific SSRL statements, in turn, are applied by calling the corresponding homonymous procedures.

The tactical natural language text generation consists in creating deep and surface tree descriptions as well as morphologically well-formed sequences of text units. The tree structures are generated in a number of steps by well-defined sequences of Lingware procedures, which represent the implemented linguistic grammar.

A dictionary-free flexing algorithm ('flexer') is applied for morphological generation. This component is also integrated into the tactical component.

3. Generation Process

The procedural rules of the described system use feature structures and unification of feature structures to determine the applicability of both phrase structure rules and feature instantiations.

The tactical generation proceeds from an initial tree structure and the sequence of SSRL statements (i.e. the input file generated by the strategic component).

The initial tree is defined as a double branching structure with a root (named S as a symbol for sentence) and two (text-) limiting sons, symbolized by $, that is, (S → $ $).

The respective sequence of SSRL statements is generated by a preceding strategic component, which is an intermediate component between the dialog manager and the tactical generation. The dialog management component pre-defines and plans the extension of system utterances and sends them to the text modelling component (the strategic component), which in turn supplements the elementary information provided by the dialog manager and determines the final sequence of SSRL statements. This sequence constitutes the basis for the tactical generation.

In terms of meaningful units of the representation, the different SSRL statements correspond to functions and program parts of the text generator. These functions initiate the generation process. The names of the representation units and the Lingware functions are identical: an SSRL statement as part of the representation at the same time represents a specific part of the generation Lingware which is called up by a procedure bearing a name which is identical to that of the SSRL statement. For instance, an SSRL statement *(action x1 "start")* calls the Lingware procedure *(action x1 "start")*, where *action* is the name of the called sub-program and *x1* and *"start"* are the arguments of the procedure.[1]

Lingware procedures matching the SSRL statements perform the first step of the generation process: they insert new nodes into a flat tree structure and assign additional information to these nodes. Thus, the first step of the tactical generation consists in creating a basic tree structure with decorated nodes corresponding to the current SSRL sequence.

When the complete sequence of SSRL statements is exhausted, that is, when all respective Lingware procedures are called and executed, a flat semantic-pragmatic tree representing the utterance to be realized has been generated. The nodes of the tree contain elementary feature value pairs. They consist of linguistic functions and relational indicators.

Further procedures are called on the subsequent generation layers in a well-defined order. They build the

[1] This *action* procedure makes sure that a node is inserted into the tree structure and puts pointers, features and values onto the node in compliance with its parameters – here, the pointer *ev-id x1* and the notion *"start"*.

structural description by starting from the deep semantic-pragmatic tree up to creating a surface-structure tree. Finally, a separate morphologic generation component is called in order to generate morphologically well-formed strings.

To sum up, during the generation process, the realizer generates flat deep structures from semantic-pragmatic expressions, then syntactic deep structures from the deep semantic-pragmatic structures, and from these syntactic deep structures surface strings. With a view to the internal process, the following steps can be distinguished.

- The first part converts semantic-pragmatic expressions into a flat deep tree structure and is totally language independent.
- The second part converts the semantic-pragmatic tree into a deep syntactic tree, which is still language independent.
- The third part converts the deep, language independent syntactic structure into a language dependent surface structure, and then from here into a well-formed text sequence of inflected words.

Thus, the most important and general type of operation performed by the text generation is to convert or transform tree structures. For this purpose the following means are employed.

- A pre-defined inventory of operators delivered by the used Lingware.
- In particular, the transformation formalism (the *xfm* operator) of the Lingware.
- The definition of elementary and specific Lingware procedures which insert sub-trees into already generated trees or modify them. (This method is comparable to the procedure of Tree Adjoining Grammars (TAGs) [14], [15].)
- The sub-tree context is defined by node decorations.
- Transformations are defined by context conditions that are composed by structure descriptions and node decorations.

4. Generation Levels

The initial generation call *g,* which has as parameter a sequence of SSRL statements, calls the procedure *generate* which has as parameter the SSRL (input) file. The procedure *generate* reads this SSRL file and calls up *xfm*, a Lingware operator with two arguments (description of the input tree structure and transformed output structure, respectively), which creates the initial tree structure and/or resets former result trees.

After creating the initial tree, *generate* calls up step-by-step the procedures described below which carry out the generation. The following generation levels are distinguished.

- Generation level 1: Generating a semantic-pragmatic deep structure. In order to complete the initial tree (i.e. S → $ $), each statement of the respective SSRL-sequence representation is evaluated. The result of this evaluation is a flat semantic-pragmatic tree. The evaluation itself is performed by the function *evals*, which has as parameter the sequence of SSRL statements.
- Generation level 2: Assigning syntactic functions and categories. Three procedures, *put-syn-func, insert-syn-cat, aux-insert,* insert elementary syntactic information into the tree structure and onto the tree nodes. These procedures have no parameters.
- Generation level 3: Application of elementary operations. After evaluating the deep input tree, elementary operations such as sub-tree insertion and manipulation (for instance, identification of coordinations, attributes and relations) are applied. The respective procedures are: *logicizer, attributor* and *relator*. They do not have parameters.
- Generation level 4: Accessing lexical and/or language specific information. After evaluating the notions/concepts supplied by the strategic component, notions/concepts are replaced with canonical forms through access to the lexicon. The lexicon access allows the application of multilingual facilities. The respective functions that perform lexicon access are *get-prep-can* and *get-lex-info*. They do not have parameters.
- Generation level 5: Evaluation of the deep-structure tree. This level generates syntactic structures (oriented towards interface structures) and takes care of gapping phenomena and insertions as language specific parts, as well as ordering of sub-clausal structures. The respective functions are: *internal-cls-structure, insert-specific-structure, del-ident-phrase* and *cls-order*. These functions do not have parameters.
- Generation level 6: Structure type realization and expansion. This level takes care of the insertion and/or transformation of pre-defined structures, determiner insertion as well as ordering of NP and AP sub-structures. The respective functions are *struct-expansion, det-insert, np-order* and *ap-order*. They do not have parameters.
- Generation level 7: Morphologic generation. At this level, the call to the morphological generation component (the 'flexer' tool) is performed. The respective function *get-inflected-form* does not have parameters.
- Generation level 8: Final refinement. At this level, phonetic refinement, pretty print and cleaning operations of the tree and the node decorations are carried out. The respective functions *corr-onset, mult-coord* and *clean-nodes* do not have parameters.
- Generation level 9: Output functions. This level provides a graphic representation of the final tree structure and output of the text string. The respective functions *draw* and *allostr* do not take any parameters.

5. Discussion

As noted earlier, we see the main strength of the described approach in the flexibility it provides to developers for controling and tuning the generation process. Often there is

more than a single way to accommodate a particular linguistic phenomenon, and as developers we need to make a design choice on which option to choose in the current context, which comprises all covered constructions of the generation task at hand. To illustrate, the phenomenon of auxiliary insertion as in English can be treated as a language specific rule, or as a parametrized application of a language independent principle. If the first option were selected, auxiliary insertion could be handled by a procedure at SAG's generation level 5 (more specifically, the *insert-specific-structure* procedure). If the second option is preferred, then a procedure *aux-insert* can be used on generation level 2, as indeed is the case in SAG. The choice made will have different ramifications for the solutions available for many other grammar phenomena that interact with auxiliary placement.

Such design choices are also familiar from declarative generation systems of realistic complexity, however in these systems they only exist on top and outside of the generation engine, which alone essentially determines the runtime behavior. This leads back to the general issue of advantages and disadvantages of declarative approaches in natural language processing, which we do not want to discuss here in detail. For our present purpose of providing a generation component operating under real-time constraints in a spoken dialog system, we have concluded that the additional control over procedural aspects provided by SAG outweighs the benefits of more restrictive declarative generation architectures in the literature.

We use SAG to build generators for multiple specific applications in a number of languages. We have lingustic expertise available to address this task. Yet, while we aim to observe generality in the range of developed generation solutions, the fine-grained procedural control has proved valuable in solving persistent problems such as over- and undergeneration, as for example in the case of resultatives in English and other languages, where the default sequence of adjective preceding head noun is reversed.

An important topic of ongoing work is a detailed comparison of resource requirements, real-time characteristics, and output quality across a number of different generation architectures. Unfortunately, by the nature of the generation task, such a comparison is rather intricate, as frequently mentioned in the literature [16].

6. Conclusions

We have described SAG, a procedural realization system for real-time applications. The procedural approach provides fine-grained control over the generation process on a sequence of nine different generation levels. Developers can therefore tune the system for a specific application depending on the observed runtime behaviour and the required range of natural language system outputs.

7. References

[1] M. Rogati, M. Walker, and O. Rambow, "Training a Sentence Planner for Spoken Dialog: The Impact of Syntactic and Planning Features", in *Proceedings of the Eurospeech 2001*.

[2] S. McRoy, S. Channarukul, and S. S. Ali, "Creating Natural Language Output For Real-time Applications", *Intelligence* 12 (2): 21-34, ACM, 2001.

[3] W.C. Mann, "An Overview of the Penman Text Generation System", *Proceedings of the Third National Conference on Artificial Intelligence*, Washington, D.C., 261-265, 1983.

[4] M. Elhadad, Using Argumentation to Control Lexical Choice: A Functional Unification-based Approach, Doctoral dissertation, Columbia University, New York, 1992.

[5] M. Elhadad, "FUF: The Universal Unifier. User manual, Version 5.2.", *Technical Report CUCS-038-91*, Columbia University, New York, 1992.

[6] K. McKeown, *Text Generation*, Cambridge University Press, Cambridge, 1985.

[7] S. Busemann, "Best-first Surface Realization", *Proceedings of the Eighth International Workshop on Natural Language Generation* 101-110, 1996.

[8] S. Busemann, and H. Horacek, "A Flexible Shallow Approach to Text Generation", *Proceedings of the Ninth International Workshop on Natural Language Generation* 238-247, 1998.

[9] K. Knight, and V. Hatzivassiloglou, "Two-level, Many-paths Generation", *Proceedings of the 33rd Annual Meeting of the ACL*, Cambridge, MA., 252-260, 1995.

[10] I. Langkilde, and K. Knight, "Generation that Exploits Corpus-based Statistical Knowledge", *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, Montreal, Canada, 704-710, 1998.

[11] I. Langkilde, and K. Knight, "The Practical Value of N-grams in Generation", *Proceedings of the Ninth International Workshop on Natural Language Generation* 248-255, 1998.

[12] D. Kallulli, "SSRL Ordering Constraints" Manuscript, SAIL LABS Technology, Vienna, 2001.

[13] J. Ritzke, "SSRL: Simple Semantic-pragmatic Representation Language", Manuscript, SAIL LABS Technology, Vienna, 2000.

[14] A.K. Joshi, "Introduction to Tree Adjoining Grammar", In A. Manaster Ramer (ed*) The Mathematics of Language*, 87-114, John Benjamins, Amsterdam, 1987.

[15] A.K. Joshi, and Y. Schabes, "Tree-Adjoning Grammars", In G. Rozenberg and A. Salomaa (eds.) *Handbook of Formal Languages*, 69-123, Springer-Verlag, Berlin, 1997.

[16] M. Galley, E. Fosler-Lussier, and A. Potamianos. "Hybrid Natural Language Generation for Spoken Dialogue Systems", in *Proceedings of the Eurospeech 2001*.