Real-Time Generation from Systemic Grammars

Terry Patten[*] and **Daniel S. Stoops**[*†]

[*]The Department of Computer and Information Science
The Ohio State University
2036 Neil Ave. Mall, Columbus, Ohio 43210

[†] AT&T Bell Laboratories
6200 E. Broad St., Columbus, Ohio 43213

Abstract

We present two compilation techniques that, when combined, enable text to be generated from systemic grammars in real-time. The first technique involves representing systemic grammars as C++ class hierarchies—this allows the inheritance in the classification hierarchies to be computed automatically by the C++ compiler at compile-time. The second technique follows from the stratified/realizational nature of systemic description that results in a mapping from semantic /contextual features to the grammar—such a mapping means that detailed grammatical features can be inferred directly, without a top-down traversal of the systemic classification hierarchies. When the mapping provides the leaf nodes of an instantiation of the grammar (as might be expected in routine generation), no traversal of the grammar is necessary whatsoever, since all the realization information from higher-level nodes has been inherited by the leaf nodes at compile-time. In such cases the text can be output in less than a second even on relatively slow workstations; on a 22 MIPS machine the run-time is too small to measure.

We have developed a framework for real-time sentence generation that we hope to deploy in future work on real-time applications. Our emphasis has been on the compilation of linguistic inference. We would like to be able to perform generation in real-time even when making adjustments for the occupation of the user, the speed of the output device (short texts for slow devices), whether or not the situation is an emergency, whether the text is spoken or written, and other situational factors that may influence linguistic decisions. A prototype implementation of our framework generates situation-adjusted clauses in less than a second on relatively slow workstations, and is too fast to measure on a 22 MIPS machine. The computational strategy behind this framework is twofold: First, we have developed an object-oriented approach to implementing systemic grammars where much of the grammatical processing is done automatically at compile-time by the C++ compiler. Second, we take advantage of stored (compiled) associations between situations and linguistic choices. Furthermore, there is an interesting synergistic relationship between these two compilation techniques.

We will first present our object-oriented implementation of systemic grammar, and provide an example of the grammatical processing. An outline of our approach to storing situation-to-language associations will then be provided. Illustrative examples will then be used to clarify these two ideas. We will then discuss the relationship between these two computational techniques, and compare our framework to other approaches to generation. Finally, some conclusions will be drawn.

An implementation of linguistic classification

Halliday's theory of *Systemic Grammar* (for a good introduction see Winograd Chapter 6) is unusual in that the primary descriptive mechanism is classification. The classification hierarchies that appear in the linguistic literature are directly analogous to those found in biology (for instance).

While linguistic classification alone may be an interesting theoretical exercise, for any practical purpose the grammar must relate these classes to linguistic structures. Just as biological classes can be related to biological properties (e.g. *mammals* have hair), linguistic classes can be related to structural properties (e.g. *declarative* clauses have subjects that precede the verb carrying the tense). Economy of description is achieved in each case because instances of a class are not only attributed the properties of that class, but also inherit the properties of all its ancestor classes. In the case of language, these properties are expressed as constraints on the structure of the clause, noun phrase, prepositional phrase or whatever is being classified. These constraints are called *realization rules* and typically refer to which constituents must appear, the order in which the constituents appear, and so on.

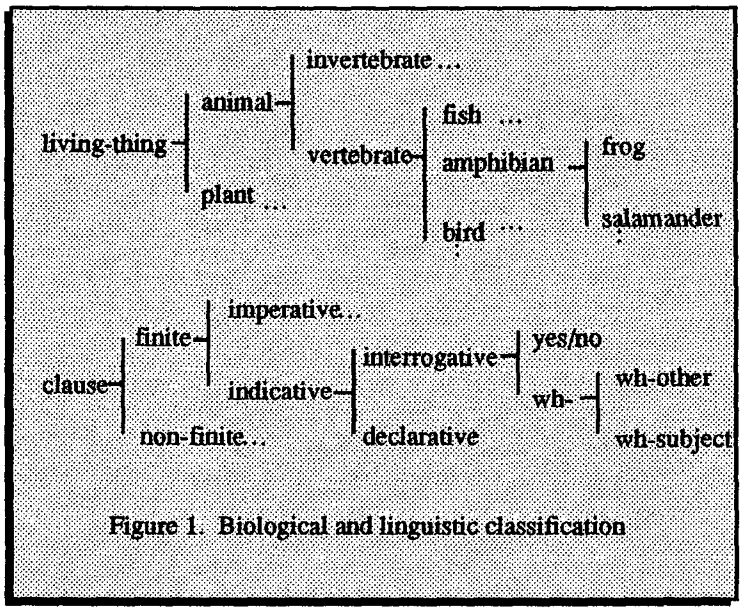

Figure 1. Biological and linguistic classification

The importance of classification hierarchies in systemic grammar led us to consider object-oriented programming as an implementation strategy (for a good discussion of some object-oriented approaches to representing linguistic knowledge, as well as a description of an object-oriented implementation of *segment grammar*, see De Smedt 90). We have chosen to explore this idea using C++. The prototype implementation is called SLANG++ (a C++ version of the Systemic Linguistic Approach to Natural-language Generation). C++ has two advantages: of primary importance for this work is that all inheritance—including multiple inheritance—is computed at compile-time; an added benefit is that C++ provides a low-overhead run-time environment. The *objects* that we are concerned with are clauses, noun phrases and so on, and each of these categories has a corresponding classification hierarchy. Systemic grammar's classification hierarchies are represented straightforwardly as hierarchies of C++ classes. The realization rules associated with each of the systemic classes are represented in a procedural form to facilitate the inheritance and construction of the appropriate English structures. After the grammar has been compiled, a leaf node in the hierarchy contains a body of code that specifies the construction of English structures according to all the realization rules associated with it and its ancestors. As we will see below, this inheritance can help to avoid traversing the grammar at run-time.

There are several steps involved in translating a systemic grammar into C++. Systemic linguists use a graphical notation that is impractical to use as input, and putting the grammar on-line is an important first step in the translation process. To this end we have used Hypercard^{TM} to create a tool that allows a systemic grammar to be entered, browsed, and modified. The card for each grammatical class shows the name, parents, children, and realization rules. Using a mouse to select parents and children, or using the keyboard to type class names, allows the user to move through the grammar to quickly find desired information. Entering a new class typically involves adding a child to an existing card, moving to the new card and entering the relevant information. The tool will not allow the

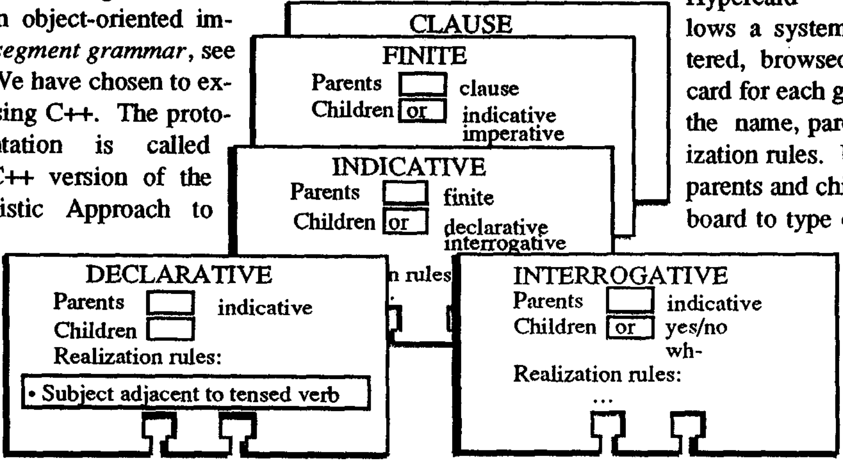

Figure 2. Simplified Hypercard representation

creation of invalid hierarchies.

The Hypercard representation of the systemic grammar is then translated, by a simple program, into C++ code. The hierarchies that were represented as links between parent cards and child cards are translated into a C++ class definition hierarchy. The (possibly multiple) inheritance in the grammar is all automatically compiled by C++ before the generation system is given any input. This means that the class description for *declarative* (for instance) will contain all the realization rules from *indicative*, *finite* and *clause* as well.

Since inheritance is computed at compile-time, more work expressed in terms of inheritance means greater run-time efficiency. If we have a text planner—or some other higher-level mechanism—that could select the leaf nodes of the classification hierarchy, then most of the grammatical processing could be done through inheritance. That is, most of the choices in the grammar would be determined by the inheritance, and would not have to be made explicitly at run-time. The problem is that the leaf nodes represent the most detailed and esoteric grammatical classes, which (as Hovy 1985, argues) should not have to be known by the higher level. In the next section we will show that this problem can be solved through the use of knowledge that associates situation classes with grammatical classes. There is no reason that such coarse-grained, compiled knowledge should not associate situations with detailed grammatical classes or even leaf nodes. In these cases the computational benefits of compiled inheritance are remarkable.

Guidance from the Situation

Our primary goal is to achieve the flexibility of natural-language even in applications where language must be processed in real time. In particular, we are interested in cases where language processing is routine, rather than the difficult special cases. McDonald, Meteer and Pustejovsky (1987) analyze the issue of efficiency in natural-language generation. They observe that:

"The greater the familiarity a speaker has with a situation, the more likely he is to have modeled it in terms of a relatively small number of situational elements which can have been already associated with linguistic counterparts, making possible the highly efficient 'select and execute' style of generation" (p. 173).

We are attempting to address the problem of how these situation-to-language associations can be stored and accessed in an efficient manner.

Halliday (1973, 1978) shows that the situation or context (including, to some extent, the information the speaker wishes to convey) can also be described using classification hierarchies. He gives an interesting theory of situation-to-language associations in his writings on "register," and some of these ideas have been discussed in the computational literature (e.g. Patten 1988a, 1988b; Bateman and Paris 1989). For our present purposes, however, it is sufficient to observe that detailed hierarchical classification schemes can be developed for situations. We represent these Hallidayan situation hierarchies using object-oriented representations in exactly the same manner as we represent the grammar. Situation classes in the hierarchy can be associated with some number of nodes in the grammatical hierarchy. Preferably these grammatical classes will be near the bottom of the hierarchy—ideally leaf nodes—because this will minimize the number of decisions that need to be made at run-time. The grammatical associations are properties of the situation classes, and are inherited at compile-time in exactly the same way as the realization rules in the grammar.

Thus, when a situation class is instantiated, the grammatical classes associated with it are then instantiated. The compile-time inheritance in the grammar ensures that all the relevant realization rules are already contained in the grammatical nodes—the grammar does not have to be traversed to locate realization rules of the ancestors. But the compile-time inheritance also avoids traversal of the situational hierarchy by passing associations down the hierarchy.

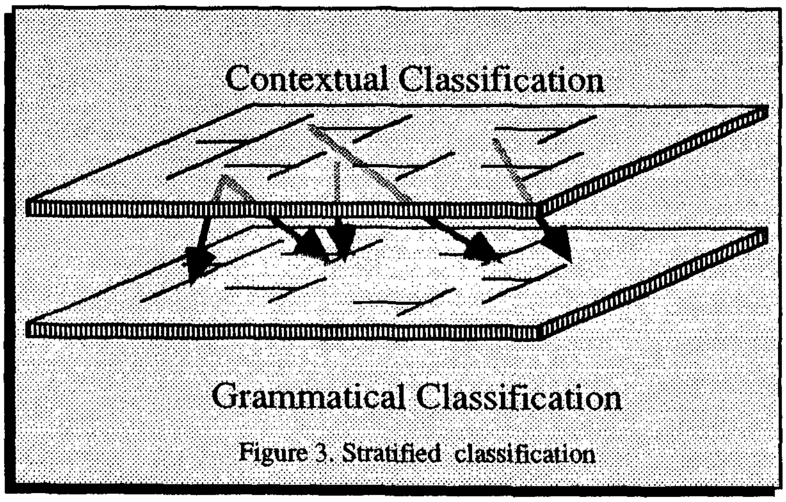

Figure 3. Stratified classification

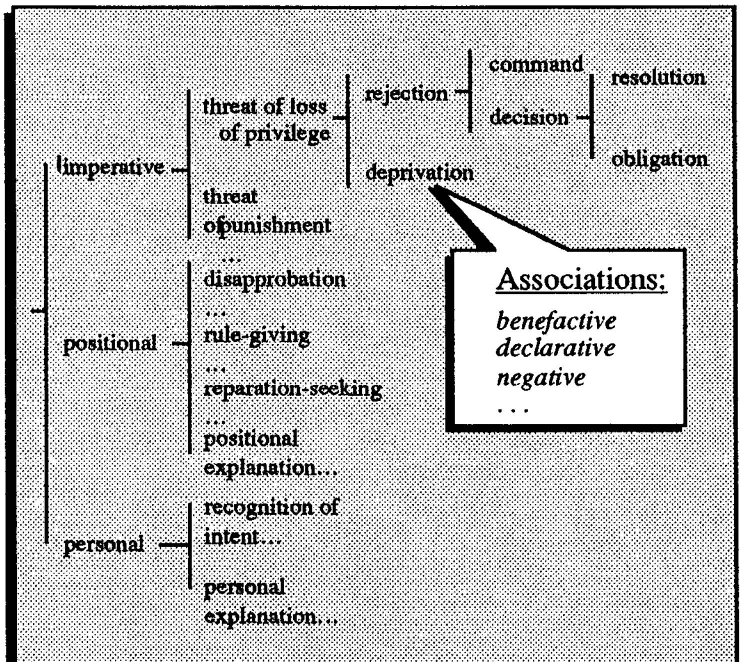

Figure 4. Choices for mother controlling child

The result is a simple and efficient transduction from situation and meaning to English structures.

Examples

The run-time operation of SLANG++ is best illustrated through examples. Our first example illustrates the processing of the grammar. Here we assume that the input to the system is a set of situational classes. That is, we assume the existence of a text planner that can pass sets of situational classes to our system—these examples are merely intended to illustrate our approach to realization. Suppose (following an example from Halliday 1978) the situation at hand involves a mother and child at home, and the child is misbehaving. The mother wants to control the behavior of the child by threatening to deprive the child of dessert. Given the situation hierarchy in Figure 4, one input to SLANG++ is the class *deprivation* (or more precisely *threat-of-deprivation*).

Several other situation classes will be input as well (indicating that they are at home and so on), but these are handled in exactly the same way as *deprivation*. Once *deprivation* has been chosen, the situation-to-language knowledge indicates that the instantiation of several grammatical classes is in order. The grammatical classes associated with deprivation include *declarative*, *benefactive*, and *negative*. Again, just looking at one of these will suffice. The representation of the class *declarative* contains not only its own realization rules (to have the subject of the clause precede the finite verb), but also all the realization rules of all its ancestors (*indicative*, *finite* and *clause*) that were inherited at compile-time. Processing these realization rules to build syntactic structures is deterministic and inexpensive (see Patten 1988a, for a detailed description of this type of realization). Other realization rules are similarly inferred from other input situational classes. Thus, in very few steps, and without any expensive search, SLANG++ computes syntactic structures (or at least structural constraints) from situational classes toward the generation of the appropriate threat (e.g. *I am not giving you a dessert*).

A second example will illustrate another important aspect of our approach—compile-time inheritance in the situation hierarchy. Sothcott (1985) describes a system that produces simple plans for building a house, does some text planning, then provides input to a sentence generator. Suppose the input is in the form of situational classes describing the building action and the role of the action in the construction process: Does it enable other actions? Is it enabled by the previous actions? Other relevant choices might include whether or not the addressee is the one responsible for this action. A simple hierarchy for this

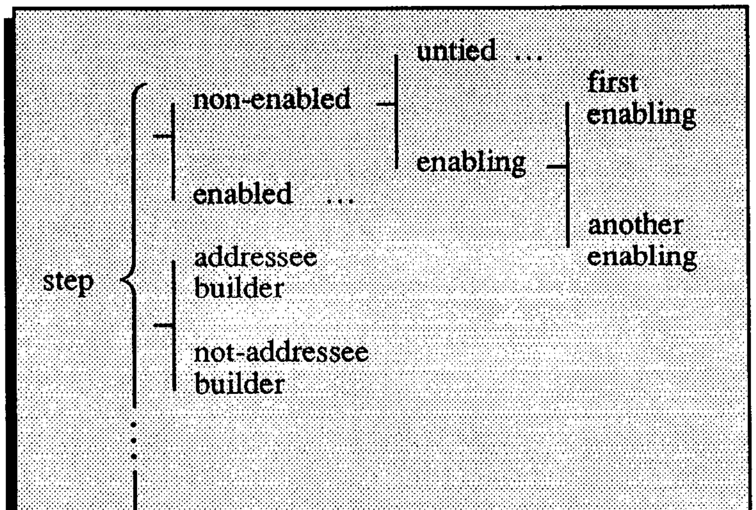

Figure 5. Some plan situations

type of situation is shown in Figure 5. Suppose we want to describe the step of sanding the floors, which has two preconditions—the carpentry and painting must both be completed. Also, suppose we are relating the plan to the person responsible for sanding and painting. The following text is generated using a clause for each action: "If the carpentry has been finished and you've done the painting, then you can sand the floors."

The generation is well illustrated by the processing required for the first clause. Since the carpentry is the first enabling action, the situational class *first-enabling* is input. Since the addressee is not responsible for the carpentry, the situational class *not-addressee-builder* is input (the situation requires *addressee-builder* for the other two clauses, resulting in the second-person subjects). The first step in the generation process is to instantiate these situational classes. The point of processing the situation hierarchy is to determine the grammatical classes associated with either the input situation classes or their ancestors. But the associations of a node's ancestors are all inherited by that node at compile-time. So, in the case where leaf situational classes are input, we are left with the much simpler problem of determining the grammatical classes associated with the input classes. If we consider the case of *first-enabling*, we see that it has inherited an association with the grammatical class *present* from its ancestor *enabling*, and the class *perfective* from its ancestor *non-enabled*. The tense of the clause is therefore immediately constrained to be present perfect. Other situational classes are instantiated resulting in further grammatical choices (including the grammatical class *declarative* discussed above) that are processed in the manner described in the previous example. Thus, the situation hierarchy benefits from compile-time inheritance just as the grammar does.

Discussion

This approach to generation consists largely of following explicit pointers into the hierarchies and collecting sets of realization rules (that have been assembled at compile time) along the way. Two types of expensive search are avoided: first, the associative knowledge avoids searching for the grammatical classes needed to satisfy situational requirements; second, the compile-time inheritance avoids traversing the grammar in search of ancestors and their realization rules. The result is an extremely efficient yet remarkably simple framework that follows naturally from combining Halliday's stratified classification with object-oriented programming.

There is an interesting synergistic relationship between the situation-linguistic associations and the compile-time inheritance. On one hand, if no associational knowledge were available, then there would be no legitimate way to access nodes near the bottom of the hierarchy, and inheritance would not be a viable operation. On the other hand, if there were no compile-time inheritance, then either a large portion of the grammar would have to be built into the associational knowledge, or an expensive grammar traversal would have to be performed at run-time. With both techniques working together, we achieve real-time performance while maintaining the desired modularity.

Our emphasis on knowledge that links situations to language has resulted in SLANG++ having much in common with the MUMBLE generator of McDonald et al.—each involves mapping from generator input to preprocessed linguistic knowledge, and each avoids an explicit traversal of the grammar. Indeed, although we use systemic grammar to represent linguistic knowledge, our system has more in common with MUMBLE than it does with other systemic generators such as Penman (Mann & Matthiessen 1983) and Proteus (Davey 1978). SLANG++ makes an important contribution to MUMBLE-style generation by demonstrating that systemic grammar can be processed in this fashion, and that the classificatory nature of systemic grammar actually enhances this approach if used in conjunction with compile-time inheritance.

Thus far we have only considered the case where the coarse-grained situation-to-language knowledge guides the generation. In practice, generation will consist of a combination of coarse-grained and fine-grained inference, with more coarse-grained inference in more familiar situations.

> "When a situation is relatively unfamiliar, its pattern of elements will tend not to have any direct mapping to natural, preconstructed structures. When this occurs, the mapping will have to be done at a finer grain, i.e. using the more abstract text properties from which preconstructed schema are built, and the process will necessarily require more effort" (McDonald et al., p. 173).

An important aspect of our object-oriented implementation is that although grammatical information is inherited through the grammar, the inherited information is only *copied* to lower nodes in the hierarchy—the information still resides in the higher-level objects and can be accessed there at run time if necessary. If a sentence is only partially specified by situation-to-language knowledge, then fine-grained linguistic knowledge must be invoked for the unresolved branches of the hierarchy. Decision specialists can be used to access the necessary information and make a choice. This technique is used in the Penman system (e.g. Mann 1985; Mann & Matthiessen 1983) as the primary strategy for processing systemic grammars. Our approach using inheritance and situation-linguistic associations improves efficiency in cases where these associations are available, but will not hamper fine-grained reasoning when it is necessary. It should therefore be possible to combine both kinds of reasoning (each of which has now been tested on large systemic grammars) to produce a system that is both efficient and robust.

Conclusion

We believe that the approach to natural-language generation that we have described here is of significant practical importance. Our C++ implementation of systemic grammar results in remarkable efficiency and simplicity. Even linguistically-sophisticated text that is tailored to the specific user and context can be generated in real-time, as demonstrated by our implementation. In cases where situation-to-language knowledge completely determines the text, we can generate appropriate sentences from a large systemic grammar in an amount of time that is too small to measure. This opens the door for natural-language generation in a wide range of time-critical applications. In cases where the situation-to-language knowledge does not completely determine the text, this knowledge can still make a significant contribution, with existing techniques for top-down hierarchy traversal able to do any remaining work. We believe that this surprisingly natural marriage of programming-language technology and linguistic theory provides a promising framework for application-oriented processing of natural language.

References

Bateman, J. A., C. L. Paris, Phrasing a text in terms the user can understand. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989, pp. 1511–1517.

Davey, A. *Discourse Production.* Edinburgh: Edinburgh University Press, 1978.

De Smedt, K. Incremental sentence generation. Technical Report 90-01, Nijmegen Institute for Cognition Research and Information Technology, 1990.

Halliday, M. A. K., *Language as Social Semiotic.* London: Edward Arnold, 1978.

Halliday, M. A. K., *Explorations in the Functions of Language.* London: Edward Arnold, 1973.

Hovy, E. Integrating text planning and production in generation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1985, pp. 848–851.

Kempen, G. (ed.), *Natural Language Generation.* Dordrecht: Nijhoff, 1987.

Mann, W. C., The anatomy of a systemic choice. In *Discourse Processes* 8, 1985, pp. 53–74.

Mann, W., C. Matthiessen, Nigel: a systemic grammar for text generation. ISI/RR-83-105, 1983.

McDonald, D., M. Meteer (Vaughan), J. Pustejovsky, Factors Contributing to Efficiency in Natural Language Generation. In G. Kempen (ed.) op. cit., 1987, pp. 159–181.

Patten, T., *Systemic Text Generation as Problem Solving.* New York: Cambridge University Press, 1988a.

Patten, T., Compiling the interface between text planning and realization. In Proceedings of the AAAI Workshop on Text Planning and Realization. 1988b, pp. 45-54.

Patten, T., G. Ritchie, A Formal Model of Systemic Grammar. In G. Kempen (ed.) op. cit., 1987, pp. 279–299.

Sothcott, C., *EXPLAN: a system for describing plans in English.* M.Sc. dissertation, Dept. of Artificial Intelligence, Univeristy of Edinburgh, 1985.

Winograd, T., *Language as a Cognitive Process, Vol. 1.* Reading, Mass.: Addison-Wesley, 1983, Chapter 6.