Computer Generation of Multiparagraph English Text[1]

William C. Mann
James A. Moore

Information Sciences Institute
University of Southern California
Marina del Rey, California 90291

This paper reports recent research into methods for creating natural language text. A new processing paradigm called Fragment-and-Compose has been created and an experimental system implemented in it. The knowledge to be expressed in text is first divided into small propositional units, which are then composed into appropriate combinations and converted into text.

KDS (Knowledge Delivery System), which embodies this paradigm, has distinct parts devoted to creation of the propositional units, to organization of the text, to prevention of excess redundancy, to creation of combinations of units, to evaluation of these combinations as potential sentences, to selection of the best among competing combinations, and to creation of the final text. The Fragment-and-Compose paradigm and the computational methods of KDS are described.

Introduction

Computer users have difficulties in understanding what knowledge is stored in their computers; the systems have corresponding difficulties in delivering their knowledge. The knowledge in the machine may be represented in an incomprehensible notation, or we may want to share the knowledge with a large group of people who lack the training to understand the computer's formal notation. For example, there are large simulation programs that get into very complicated states we would like to be able to understand easily. There are data base systems with complex knowledge buried in them, but real problems in extracting it. There are status-keeping systems from which we would like to get snapshots. There are systems that try to prove things, from which we would like to have progress reports and justifications for various actions. Many other kinds of systems have knowledge-delivery difficulties.

The circumstances that make it particularly attractive to deliver this knowledge in natural language are: a) complexity of the source knowledge, so that its notation is not easily learned, b) unpredictability of the demands for knowledge, so that the actual demands cannot be met with specific preprogrammed output, and c) the need to serve a large pool of untrained or lightly trained users of these systems.

For a number of the kinds of systems mentioned above, getting the information out is one of the principal limitations on the systems' uses. If the information could be accessed more easily, then far more people could use the systems. So we are talking in part about facilitating existing systems, but much more about creating new opportunities for systems to serve people.

If computer systems could express themselves in fluent natural language, many of these difficulties would disappear. However, the necessary processes for such expression do not exist, and there are formidable obstacles even to designing such processes. The theory of writing is sketchy and vague, and there are few interesting computer systems to serve as precedents. Any research effort to create such systems — systems that know how to write — can be significant both in its practical implications and for the knowledge of writing that it produces.

[1] This research was supported in part by National Science Foundation grant No. MCS76-07332 and in part by the Air Force Office of Scientific Research contract No. F49620-79-c-0181. The participation of Neil Goldman and James Levin is gratefully acknowledged. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research of the U.S. Government.

0362-613X/81/010017-13$01.00

Writing is an intellectually interesting task, though poorly understood. If we want to have a better theory, a better characterization of this task, then we can use computer program design and test as a *discovery procedure* for exploring the subject. In the present state of the art, the same research can create both theoretical knowledge and practical computational methods.

Of course, in a limited sense, programs already deliver knowledge in natural language by using "canned text." A person writes some text, possibly with the use of blank segments, and the text is stored for use in association with a particular demand. The machine fills in the blanks as needed in a way anticipated as sufficient for the demand. This is a very useful technique, but it does not tell us much about the task of writing, and it does not generalize to situations in which the need for text has not been well anticipated in the programming stage. The basic organization and writing of canned text are done by a person, so that sort of "writing" falls outside the scope of this paper.

The idea that natural language would be a useful output medium for a computer system is quite old, but the research to make such output possible is still in a very primitive state. Most of it has concentrated on sentence generation, in which problems such as word selection or translation to natural language from logical formulas have been chosen for study. These are important, but the needs for knowledge delivery simply cannot be satisfied by single sentences. It is necessary to have multiple sentences to communicate complex information, and multiple paragraphs would be even more valuable.

The remainder of this paper thus focuses on computer generation of multisentence text. We present some examples of multisentence output from past systems and then discuss our own research.

Multisentential Text Generation in Past Research

Several systems have generated multisentence text [2,5,8,10,11].[2] [3] Sample texts from three of these are shown in Figure 1. One example is the SCHOLAR system, which describes aspects of South American geography. SCHOLAR's description of a South American country was several sentences long [2]. The SAM system, a story generator operating from a conceptual dependency script, also created multisentential output [10]. More recently, Meehan developed TALE-SPIN, a fable generator that worked in a different way, also based on a conceptual dependency representation [8].

SCHOLAR:

TELL ME ABOUT THE LOCATION OF URUGUAY.

THE LOCATION IN URUGUAY IS SOUTH AMERICA.
THE LATITUDE RANGES FROM -30 TO -35 DEGREES.
THE LONGITUDE RANGES FROM -53 TO -58 DEGREES.
THE NORTHERN AND EASTERN BORDERING COUNTRY IS BRAZIL.
THE WESTERN BORDERING COUNTRY IS ARGENTINA.
THE BOUNDARY IS THE URUGUAY RIVER.
THE SOUTHEASTERN COAST IS THE ATLANTIC OCEAN.
THE SOUTHERN COAST IS THE RIO DE LA PLATA.

SAM:

John decided he was going to a restaurant. He went to one. He asked the hostess for a table. She told him he could go to it. He went to it. She went to it. He sat down in a chair. He got a menu from her. He read it. A waiter saw John was at the table. The waiter went to it. John ordered a lobster. A chef cooked it. The waiter got it from the chef. The waiter served it to John quickly. He ate it. He asked the waiter for a check. John got it from the waiter. John remembered the waiter had served the lobster to him quickly. John left the waiter a large tip. John paid the check. He left the restaurant.

TALESPIN:

ONCE UPON A TIME GEORGE ANT LIVED NEAR A PATCH OF GROUND. THERE WAS A NEST IN AN ASH TREE. WILMA BIRD LIVED IN THE NEST. THERE WAS SOME WATER IN A RIVER. WILMA KNEW THAT THE WATER WAS IN THE RIVER. GEORGE KNEW THAT THE WATER WAS IN THE RIVER. ONE DAY WILMA WAS VERY THIRSTY. WILMA WANTED TO GET NEAR SOME WATER. WILMA FLEW FROM HER NEST ACROSS A MEADOW THROUGH A VALLEY TO THE RIVER. WILMA DRANK THE WATER. WILMA WASN'T THIRSTY ANY MORE.

GEORGE WAS VERY THIRSTY. GEORGE WANTED TO GET NEAR SOME WATER. GEORGE WALKED FROM HIS PATCH OF GROUND ACROSS THE MEADOW THROUGH THE VALLEY TO A RIVER BANK. GEORGE FELL INTO THE WATER. GEORGE WANTED TO GET NEAR THE VALLEY. GEORGE COULDN'T GET NEAR THE VALLEY. GEORGE WANTED TO GET NEAR THE MEADOW. GEORGE COULDN'T GET NEAR THE MEADOW. WILMA WANTED TO GET NEAR GEORGE. WILMA GRABBED GEORGE WITH HER CLAW. WILMA TOOK GEORGE FROM THE RIVER THROUGH THE VALLEY TO THE MEADOW. GEORGE WAS DEVOTED TO WILMA. GEORGE OWED EVERYTHING TO WILMA. WILMA LET GO OF GEORGE. GEORGE FELL TO THE MEADOW. THE END.

Figure 1. Some published multisentence text samples.

These systems share several features. First, the data structures that are the basis of the generation were designed for text processing; many of the special demands of text processing were anticipated and accommodated in the design of the knowledge structures themselves. Second, the sentence boundaries in these systems were direct correlates of internal features of

[2] Since this paper was written in 1979, additional work involving multisentential generation has been done or published [3,4,7]. Only the references have been updated.

[3] All of the cited research has worked with English, but other languages would have served as well.

the data structures themselves. Often the sentence order arose in the same way.[4] Third, these systems had fixed generation goals, implicit in the code. Thus, the reader's needs were taken to be fixed and preknown by the system. Fourth, although goal-pursuit could sometimes be described in the material being generated, the systems themselves did not operate on a goal-pursuit algorithm. Finally, none of these systems chose the particular sentences to use in their output on the bases of quality assessment or comparisons among alternatives.

In all five of these points, the KDS research contrasts with these previous efforts. We have worked with data structures not designed for text generation; the sentence boundaries we develop are not direct correlates of internal features of the data structures; there are explicit goals for the generation process to satisfy; the system itself pursues goals; and the final text is chosen through quality comparisons among alternative ways of saying things.

The Task for the Knowledge Delivery System

In the light of these considerations, the problem can be restated more specifically as follows:

Given

1. An **explicit goal** of knowledge expression,
2. A computer-internal **knowledge base** adequate for some non-text purpose, and
3. **Identification of the parts** of the knowledge base that are relevant to the goal,

the task is to produce clean, **multiparagraph text**, in English, which satisfies the goal.

The Partitioning Paradigm

When we have stated this task to AI workers familiar with natural language processing, with no further specification, they have expected a particular kind of solution. They say, "Well, there are some sentence generators around, but the given information structures are too large to be expressed in single sentences. Therefore what we need is a *method for dividing up the input structure into sentence-size pieces.* Then we can give the pieces to a suitable sentence generator and get the desired text." This is the expected solution, and people will simply presume that it is the line of development being taken.

That approach, which we call the Partitioning paradigm for text generation, was used in all the systems described above. For the Partitioning paradigm to work, the generation task must be simplified by features of the knowledge base:

1. The knowledge base data structures have features that indicate appropriate sentence boundaries, and
2. The pieces of information appropriate to be expressed in an individual sentence are adjacent. That is, a process can access all of the information appropriate to be expressed in a single sentence by following the data structure, without being required to traverse information to be expressed in other sentences.

These conditions prevail (by design) in all of the systems described above, but they are not generally typical of information storage in computers. As we will see, KDS takes an entirely different approach to the problem.

Several inherent difficulties become apparent when we attempt to use partitioning:

1. **Missing adjacencies** — Since (by our problem definition) the knowledge comes from a structure not prestructured for the generation task, what is and what is not adjacent in the knowledge base may be quite arbitrary. We may wish to include several widely scattered items in a sentence, so that it is not possible to carve out a piece with those items in it at all. The adjacencies that we need in order to partition the structure into sentence-size parts may simply be absent.

2. **Intractable residues** — Even though we may be able to find some way to **start** cutting out sentence-size objects from the data structure, there is no assurance at all that we will be able to run that method to completion and carve the *entire* structure into sentence-size pieces. Think of the comparable problem of carving statues from a block of marble. We may be able to get one statue or several, but if every part of the original block must end up looking like a statue, ordinary carving methods are insufficient. The residues left after carving out the first few statues may be intractable. A comparable sort of thing can happen in attempting to partition data structures.

3. **Lack of boundary correlates** — In some ways the worst difficulty is that an arbitrary given data structure does not contain structural correlates of good sentence boundaries.

[4] This is not to say that sentence boundaries are always one for one with data structures, nor that the data structures always contain all the information used in making a sentence. But the forms of data structures in these systems have been shaped almost exclusively by natural language processing tasks, which tends to make sentence boundary determination easy. The content of those structures has often been filled in manually, leaving indeterminable the relative contributions of program and programmer.

> One cannot inspect the data structure and tell in any way where the sentence boundaries ought to be. Along with the other difficulties, this has led us to reject the expected solution, the Partitioning paradigm, and to create another.

The Fragment-and-Compose Paradigm

Our solution comes in two steps:

1. Find methods for *fragmenting the given data structure into little pieces,* small propositional parts.
2. Find methods for *composing good sentences and good paragraphs out of those little parts.*

We call this the Fragment-and-Compose paradigm. It is interesting to note that other systems employ a Fragment-and-Compose approach — e.g., building construction, papermaking, and digestion. In each, one begins by producing small, easily manipulated objects much smaller than the desired end-product structures, and then assembles these into the desired end products in a planned, multistage way. For the block of marble, the comparable processes are crushing and casting.

We may not be very encouraged in our text generation task by such precedents. However, there are precedents much closer to our actual task. The task of natural language translation resembles in many ways the task of translating from a computational knowledge source (although it has a comprehension subtask which we lack). Consider the *(annotated)* quotation below from *Toward a Science of Translating* [9].

> The process by which one determines equivalence (*faithfully translates*) between source and receptor languages is obviously a highly complex one. However, it may be reduced to two quite simple procedures:
>
> (1) "decomposition" of the message into the simplest semantic structure, with the most explicit statement of relationships; and
>
> (2) "recomposition" of the message into the receptor language.

The quotation is from Nida's chapter on translation procedures. Notice particularly the two steps: *decomposition* and *recomposition*, and the emphasis on *simple, explicit semantic structures* in the results of the decomposition.

It turns out that this is the central procedural statement of Nida's book, and the remainder of the book can be seen as giving constraints and considerations on how this decomposition and recomposition ought to take place. We have very good reasons here for expecting that Fragment-and-Compose is an appropriate paradigm for natural language knowledge delivery.

To give a sense of what can be done using Fragment-and-Compose, here is a piece of a machine-generated text (created by KDS) about what happens when fire breaks out in the computer room.

```
Whenever there is a fire, the alarm system is
started, which sounds a bell and starts a timer.
Ninety seconds after the timer starts, unless the
alarm system is cancelled, the system calls Wells
Fargo.  When Wells Fargo is called, they, in
turn, call the Fire Department.
```

Description of KDS

Figure 2 is a block diagram of KDS, which simply says that KDS takes in an Expressive Goal (telling what the text should accomplish relative to its reader) and also a pre-identified body of Relevant Knowledge in the notation of its source. The output is multiparagraph text that is expected to satisfy the goal.

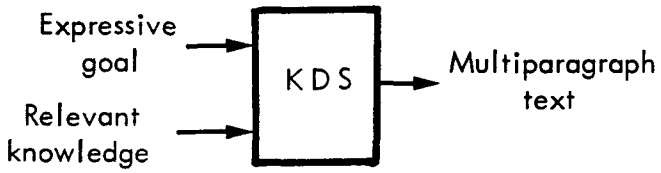

Figure 2. Input and output of KDS.

We will be carrying a single example through this description of KDS. It is the most complex example handled by KDS, and it incorporates many ideas from previous studies on description of computer message systems.

A small contingency-plans data base contains knowledge about what happens in various circumstances, and about people's actions, responsibilities, authorities, and resources. The particular knowledge to be delivered concerns a computer room in which there may be some indication of fire and in which there is a computer operator who should know what to do if that happens. This operator is the nominal reader of the text.

The general Expressive Goal is that the computer operator will know what to do in all of the predictable contingencies that can arise starting with an indication of fire. The contingencies are represented in the "Fire Alarm Scene," part of the knowledge base. A schematic sketch of the Fire Alarm Scene is given in Figure 3. (The figure is expository and contains far less information than the actual Scene. The Scene is a "semantic net," a collection of LISP expressions that refer to the same objects.)

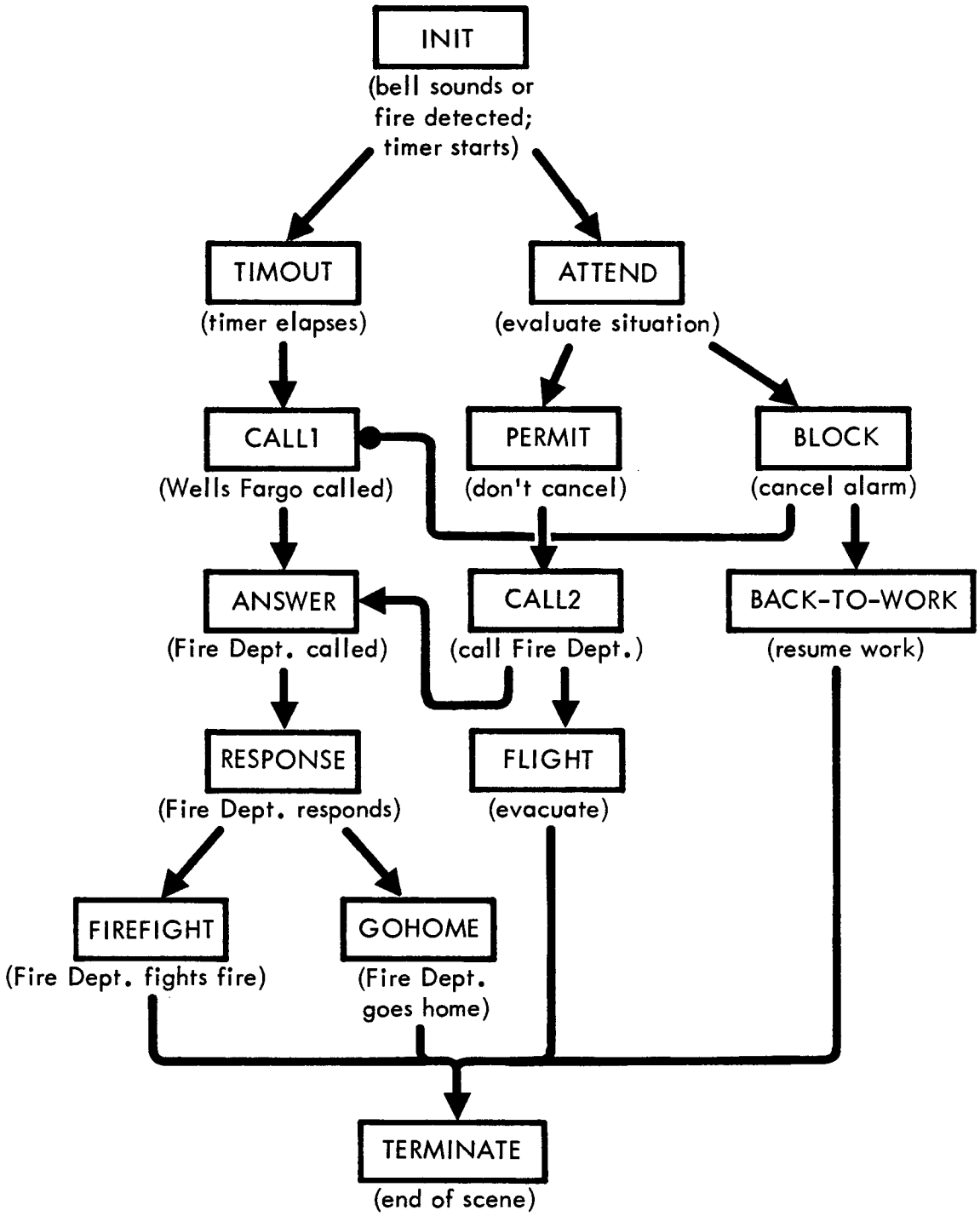

Figure 3. Events in the Fire-Alarm scene.

KDS MODULES	MODULE RESPONSIBILITIES
FRAGMENTER	• Extraction of knowledge from external notation • Division into expressible clauses
PROBLEM SOLVER	• Style selection • Gross organization of text
KNOWLEDGE FILTER	• Cognitive redundancy removal
HILL CLIMBER	• Composition of concepts • Sentence quality seeking
SURFACE SENTENCE MAKER	• Final text creation

Figure 4. KDS module responsibilities.

The knowledge identified as relevant includes not only the events of this scene but also enough information to support another computational task. In this example the knowledge is sufficient to support an alternate task, which we call the Motivation Exhibit task, i.e., to exhibit, for each action in the scene, the actor's reasons for performing the action. So, for example, the relevant knowledge includes the knowledge that fires destroy property, that destroying property is bad, that the amount of property destroyed increases with the duration of the fire, and that the Fire Department is able to employ methods for stopping fires. This is sufficient to be able to explain why the Fire Department attempts to stop fires. KDS does not perform the Motivation Exhibit task, but its knowledge is sufficient for it. We generate from a knowledge base sufficient for multiple tasks in order to explore the problems created when the knowledge representation is not designed for text processing.

The content of the scene is as follows:

> In the beginning state, INIT, the fire alarm sounds a bell. As we follow down the left side of the figure, we see that the fire alarm starts an interval timer, and at the end of the interval, the timer automatically phones Wells Fargo Company, the alarm system manager. Wells Fargo phones the Fire Department, and the Fire Department comes. The Fire Department fights the fire if there is one, and otherwise goes home.
>
> Meanwhile, the computer operator must pay attention to the alarm and decide what to do. He can block the alarm system's action, cancelling the alarm, or he can let the alarm system take its course. In the latter case, his next duty is to call the Fire Department himself, which has the same effect as Wells Fargo calling it. After that, his next duty is to flee. If he blocks the alarm then he is to go back to his previous task.

Major Modules of KDS

KDS consists of five major modules, as indicated in Figure 4. A Fragmenter is responsible for extracting the relevant knowledge from the notation given to it and dividing that knowledge into small expressible units, which we call fragments or protosentences. A Problem Solver, a goal-pursuit engine in the AI tradition, is responsible for selecting the presentational style of the text and also for imposing the gross organization onto the text according to that style. A Knowledge Filter removes protosentences that need not be expressed because they would be redundant to the reader.

The largest and most interesting module is the Hill Climber, which has three responsibilities: to compose

complex protosentences from simple ones, to judge relative quality among the units resulting from composition, and to repeatedly improve the set of protosentences on the basis of those judgments so that it is of the highest overall quality. Finally, a very simple Surface Sentence Maker creates the sentences of the final text out of protosentences.

The data flow of these modules can be thought of as a simple pipeline, each module processing the relevant knowledge in turn. We will describe each of these modules individually.

Fragmenter Module

The Fragmenter (Figure 5) takes in the relevant knowledge as it exists externally and produces a set of independent protosentences, called the Sayset. These primitive fragments, the protosentences, have no intended order. (In our final tests, they are presented in a list that is immediately randomized.) Each primitive protosentence can, if necessary, be expressed by an English sentence.

Figure 5. Fragmenter module input and output.

To help the reader understand the level of these fragments, were they to be expressed in English, they would look like:

"Fire destroys objects."
"Fire causes death."
"Death is bad."
"Destroying objects is bad." etc.

So the problem for the remainder of the system is to express well what can surely be expressed badly. It is important to note that this is an improvement problem rather than a problem of making expression in English feasible.

The protosentences the Fragmenter produces are propositional and typically carry much less information than a sentence of smooth English text. In our example, the fragmenter produces the list structures shown in part below for two of its fragments.

```
((CONSTIT (WHEN (CALLS NIL WELLS-FARGO)
                (CALLS WELLS-FARGO FIRE-DEPT)))...)
((CONSTIT (WHENEVER (STARTS NIL ALARM-SYSTEM)
                (PROB (SOUNDS ALARM-SYSTEM BELL)...)
```

These fragments encode: "When {unspecified} calls Wells Fargo, Wells Fargo calls the Fire Department." and "Whenever {unspecified} starts the alarm system, the alarm system probably sounds the bell."

Problem Solver Module

The second major module is the Problem Solver (Figure 6). The primary responsibilities of the Problem Solver are to *select a text presentation style* and to *organize the text content according to the selected style.* For this purpose, it has a built-in taxonomy of styles from which it selects. Although the taxonomy and selection processes are very rudimentary in this particular system, they are significant as representatives of the kinds of structures needed for style selection and style imposition.

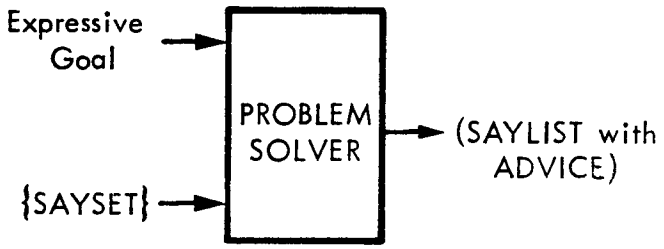

Figure 6. Problem Solver input and output.

We believe that text style should be selected on the basis of the *expected effects.* In simple cases this is so obvious as to go unrecognized; in more complex cases, which correspond to complex texts, there are many stylistic choices. In order to select a style, one needs:

1. A description of the effect the text should have on the reader,
2. Knowledge of how to apply stylistic choices, and
3. A description of the effects to be expected from each stylistic choice.

Note that these are required whether stylistic choices are distributed or holistic, i.e., whether they are made in terms of attributes of the final text or in terms of particular methods for creating or organizing the text.

The first requirement above, a description of desired effects, is (more or less by definition) a *goal.* The second item is the set of applicable methods, and the third is the knowledge of their effects. The Problem Solver is a goal-pursuit process that performs means-ends analysis in a manner long familiar in AI. The information organization is significant partly because of the demand it puts on the knowledge of style: *Knowledge of style must be organized according to expected effect.* Otherwise, the program has no adequate basis for selecting style.

The Problem Solver takes in the Sayset produced by the Fragmenter and the Expressive Goal given to the system and produces a Saylist, which is an ordered list of the protosentences, some of which have been marked with Advice. The Problem Solver pursues given goals. It has several submodules that specialize in particular kinds of goals, including modules Tell and Instructional-narrate, which are active on this example. The Problem Solver can operate on the current Saylist with three kinds of actions in any of its modules:

1. It can Factor the Saylist into two sublists separated by a paragraph break. It extracts all protosentences with a particular character or attribute and places them above the paragraph break, which is above all those that lack that attribute. Order within each sublist is retained.
2. It can impose an order on some or all of the elements of the Saylist.
3. It can mark protosentences with Advice. Sometimes the Problem Solver knows some attribute of the final text that ought to be achieved, perhaps because of a demand of the chosen style, but it has no way to effect this directly. In this case it marks all the affected protosentences with Advice, which will be acted on after the Problem Solver has finished.

Figure 7 describes the rules used in the Problem Solver that carry out these three kinds of actions. In this example, the Tell module acts before Instructional-narrate. The Factoring rules are applied sequentially, so that the last one prevails over previous ones.

The first Tell rule corresponds to the heuristic that the existence of something ought to be mentioned before its involvement with other things is described. The third rule corresponds to the heuristic that the writer (KDS) ought to reveal its own goals of writing before pursuing those goals.

Instructional-narrate uses a presentational technique that makes the reader a participant in the text. So, for example, the final text says, "When you hear the alarm bell ...," rather than "When the operator hears the alarm bell...," Instructional-narrate knows that the role of "you" should be emphasized in the final text, but it has no direct way to achieve this. To every protosentence that refers to "you," it attaches advice saying that explicit reference to the reader, which is done by mentioning "you" in the final text, has positive value. This advice is taken inside the Hill-climber.

In our example the Problem Solver creates the following fragment:

```
(PARAGRAPH-BREAK (REASON: (BOUNDARY NON-H-ACTOR)))
((CONSTIT (WHEN (IF (POSSIBLE)
                    (CALL YOU FIRE-DEPT))
                (EVOKE YOU EVAC-SCENE)))...
 (ADVISORS FRAG INST-NARRATE)
 (ADVICE ...(GOOD YOU)))
```

These represent: "(Put a paragraph break here because the actions of agents other than the hearer end here)" and "If possible, call the Fire Department; then, in either case, evacuate. (Advised by FRAG and INST-NARRATE Modules) (Advised that YOU is GOOD)".

Knowledge Filter Module

The Knowledge Filter is a necessary part of KDS because as soon as we attempt to create text from a knowledge base suitable to support some other computational purpose, we find a great deal of information there that ought not to be expressed, because the reader already knows it.

This is a general phenomenon that will be encountered whenever we generate from an ordinary computational knowledge base. As an illustration, consider Badler's work on getting a program to describe a movie in English.

Factoring Rules:

TELL

1. Place all (EXISTS ...) propositions in an upper section.
2. Place all propositions involving anyone's goals in an upper section.
3. Place all propositions involving the author's goals in an upper section.

INSTRUCTIONAL-NARRATE

1. Place all propositions with non-reader actor in an upper section.
2. Place all time dependent propositions in a lower section.

Ordering Rules:

INSTRUCTIONAL-NARRATE

1. Order time-dependent propositions according to the (NEXT ...) propositions.

Advice-giving Rules:

INSTRUCTIONAL-NARRATE

1. YOU is a good thing to make explicit in the text.

Figure 7. Rules used in the Problem Solver.

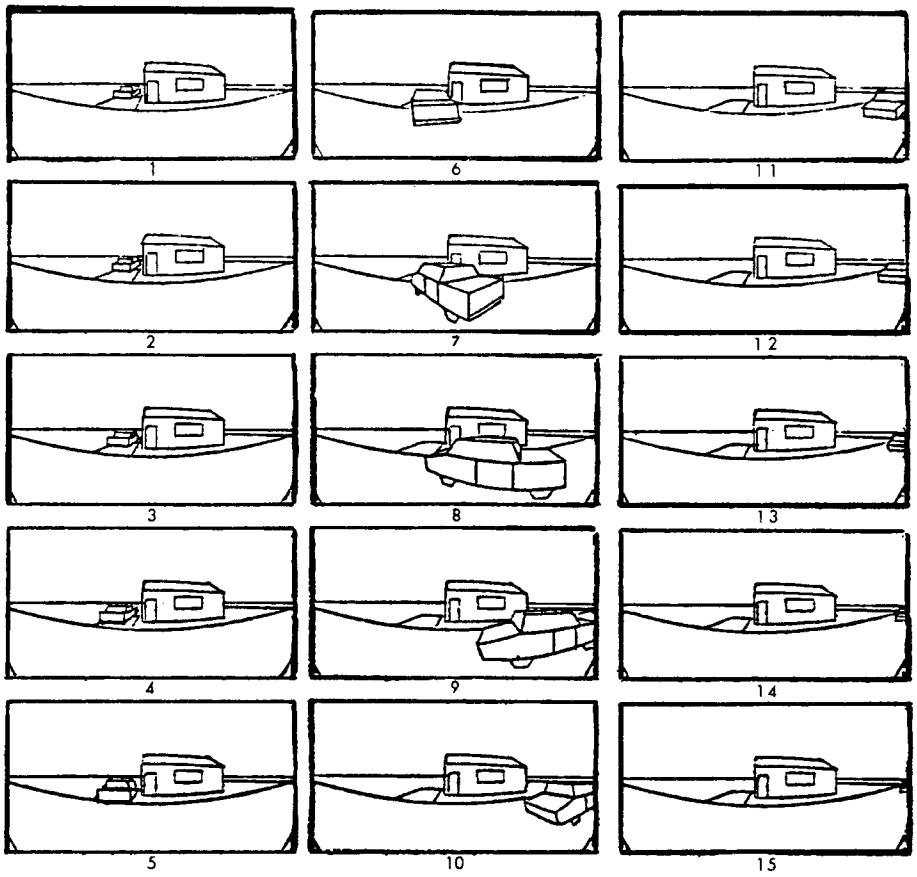

Figure 8. Badler's "Moving Car Scenario".

Figure 8 is reproduced from [1]. It shows fifteen successive scenes from a short computer-generated movie. The graphics system that generates the movie provides a stock of propositional knowledge about it. The objects in the scene are known to the machine unambiguously and in sufficient detail to generate the movie. The research task is to create a computer program that will describe in English the physical activities in this and similar movies. The detail is voluminous, and so Badler is faced with a serious information suppression problem. After several stages of applying various filtering heuristics, such as "Don't describe directly anything that doesn't move," he can represent the movie by the five statements below.

1. There is a car.
2. The car starts moving toward the observer and eastward, then onto the road.
3. The car, while going forward, starts turning, moves toward the observer and eastward, then northward-and-eastward, then from the driveway and out-of the driveway, then off-of the driveway.
4. The car, while going forward, moves northward-and-eastward, then northward, then around the house and away-from the driveway, then away-from the house and stops turning.
5. The car, while going forward, moves northward, then away.

These are still too cumbersome, so additional stages of reduction are applied, yielding the single statement:

> The car approaches, then moves onto the road, then leaves the driveway, then turns around the house, then drives away from the house, then stops turning, then drives away.

Even the longer text above contains only a fraction of the available information about the car and the other objects. Information on their types, their subparts, visibility, mobility, location, orientation and size are available from Badler's source. He also develops a sequence of events to describe the movie, based on certain indicators of continuity and discontinuity. *The volume of information available, the predictability of its parts, and the insignificance of some of its details are such that all of it could not have been expressed in a smooth text.*

One of the principal activities of Badler's system is selection of information to be removed from the set of ideas to be expressed. Some things need not be expressed because they follow from the reader's general knowledge about motion of objects; others are removed because they represent noise, rather than significant events, generated by the processes that discern motion.

The point for us is simply that *the demands of smooth text production are incompatible with expression of all of the available information.* Text production requires condensation and selectivity, the process we call knowledge filtering, on any reasonably complete body of knowledge. Knowledge filtering is a significant intellectual task. It requires coordinated use of a diversity of knowledge about the reader, the knowledge to be delivered, and the world in which all reside. We now recognize the necessity of sophisticated knowledge filtering as part of the process of producing quality text.

KDS's Knowledge Filter (Figure 9) inputs the Saylist, including Advice, from the Problem Solver, and outputs the Saylist with additional Advice, called "Don't Express" advice, on some of the protosentences. So some of the items have been marked for omission from the final text. (They are marked rather than deleted so that they are available for use if needed as transitional material or to otherwise make the resulting text coherent.) The knowledge filter decides which protosentences to mark by consulting its internal model of the reader to see whether the propositional content is known or obvious. The model of the reader, in this implementation, is very simple: a collection of propositions believed to be known by him. Although KDS's reader model does not contain any inference capabilities about what is obvious, a more robust model certainly would. We recognize that the work of the Knowledge Filter is a serious intellectual task, and we expect that such a filter will be an identifiable part of future text creation programs.

In our example the Knowledge Filter produces the DON'T-EXPRESS advice in the following element of the Saylist:

```
((CONSTIT (WHENEVER (SOUNDS NIL ALARM-BELL)
                    (HEARS YOU ALARM-BELL)
                    (PROB)))...
 (ADVISORS INST-NARRATE NONEXP)
 (ADVICE (GOOD YOU)
         DON'T-EXPRESS))
```

In this case, the involvement of the reader in (HEARS YOU ALARM-BELL) arises from the Advice-giving rule for Instructional-Narrate. It indicates that it is good to express this. The DON'T-EXPRESS arises from the Knowledge Filter, indicating that it is unnecessary to express this. DON'T-EXPRESS prevails.

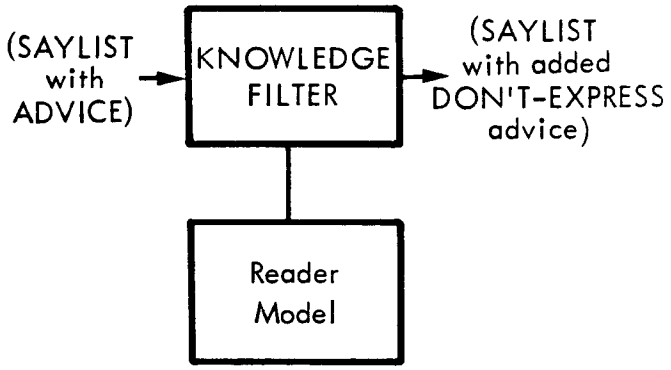

Figure 9. Knowledge Filter module input and output.

Hill Climber Module

The Hill Climber module (Figure 10) consists of three parts:

1. A somewhat unconventional hill-climbing algorithm that repeatedly selects which one of an available set of changes to make on the Saylist.
2. A set of Aggregation rules (with an interpreter) telling how the protosentences may legally be combined. These correspond roughly to the clause-combining rules of English, and the collection represents something similar to the writer's competence at clause coordination. Each Aggregation rule consumes one or more protosentences and produces one protosentence. Advice propagates onto the protosentences produced.
3. A set of Preference rules (with an interpreter) able to assign a numerical quality score to any protosentence. The score computation is sensitive to Advice.

The algorithm is equivalent to the following: Scores are assigned to all of the primitive protosentences; then the Aggregation rules are applied to the Saylist in all possible ways to generate potential next steps up the hill. The resultant protosentences are also evaluated, and the Hill Climber algorithm then compares the scores of units consumed and produced and calculates a net gain or loss for each potential application of an Aggregation rule. The best one is executed, which means that the consumed units are removed from the Saylist, and the new unit is added (in one of the positions vacated, which one being specified in the Aggregation rule).

This process is applied repeatedly until improvement ceases. The output of the Hill Climber is a Say-

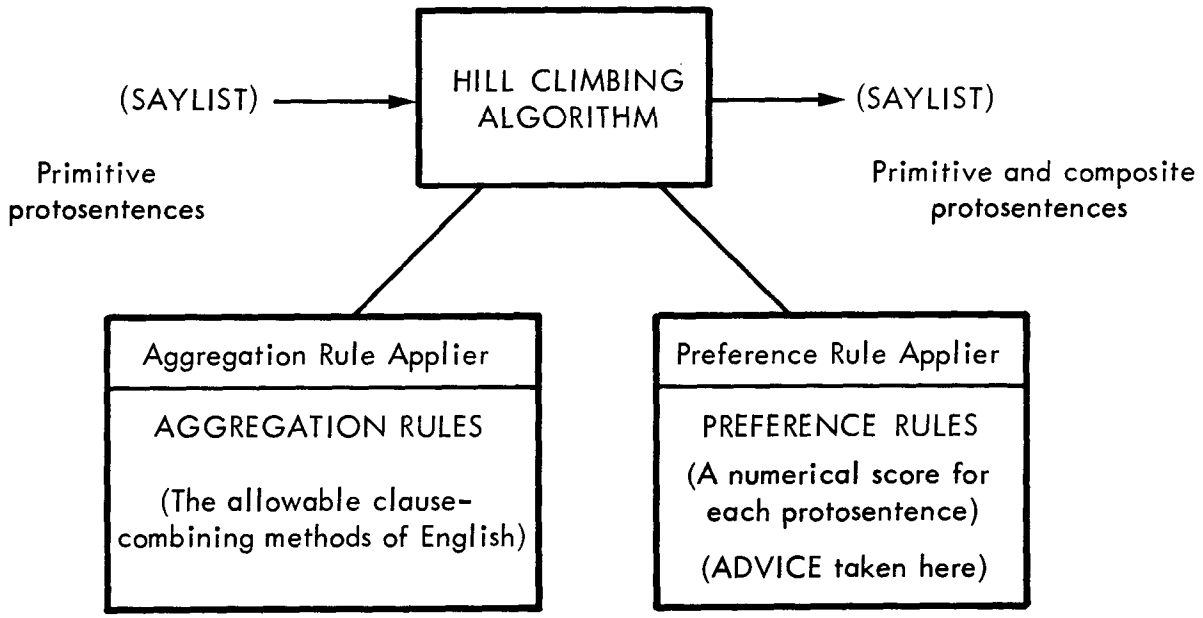

Figure 10. Hill Climber module.

list for which there are no remaining beneficial potential applications of Aggregation rules.

The selection algorithm of the Hill Climber is somewhat unconventional in that it does not select the Aggregation rule application with the largest increase in collective score, which would be the usual practice. The hill of collective scores has many local maxima, which can be traced to the fact that one application of an aggregation rule will preclude several others. Because protosentences are consumed, the various applications are in competition, and so a rule that produces a large gain may preclude even more gain.

The Hill Climber selects the rule application to use based on an equation that includes competitive terms. It computes the amount of gain *surely precluded* by each application and makes its selection on the basis of maximum net gain, with the precluded gain subtracted.

The use of hill climbing avoids the combinatorial explosion involved in searching for the best of all possible ways to express the content. In general only a tiny fraction of the possibilities are actually examined.

This Saylist improvement activity is the technical heart of the text production process; it develops the final sentence boundaries and establishes the smoothness of the text.

Figure 11 shows a few of the Aggregation rules. (Each of them has been rewritten into an informal notation suggesting its content.) Aggregation rules are intended to be meaning-preserving in the reader's

1. COMMON CAUSE.
   Whenever C then X.
   Whenever C then Y.
   } Whenever C then X and Y.
2. CONJOIN MID-STATE
   Whenever X then Y.
   Whenever Y then Z.
   } Whenever X then Y and then Z.
3. DELETE MID-STATE
   Whenever X then Y.
   Whenever Y then Z.
   } Whenever X then Z.
4. DELETE EXISTENTIAL
   There is a Y.
   <mention of Y>
   (Y is known unique)
   } <mention of Y>
5. IF-THEN-ELSE
   If P then Q.
   If not P then R.
   } If P then Q otherwise R.
6. TEST AND BRANCH
   When P then determine if X.
   If X then Q.
   If not X then R.
   } When P then determine X and decide Q or R.

Figure 11. Sample Aggregation rules.

comprehension, but are not intended to preserve explicitness.

These are only a few of the Aggregation rules that have been used in KDS; others have been developed in

the course of working on this and other examples. Coverage of English is still very sparse. In other examples, an aggregation rule has been used to produce a multiple-sentence structure with intersentential dependencies.

Figure 12 shows the Preference rules. They were derived empirically, to correspond to those used by the author of some comparable human-produced text.

1. Every protosentence gets an initial value of -1000.
2. Every primitive protosentence embedded in a composite protosentence decreases value by 10.
3. If there is advice that a term is good, each occurrence of that term increases value by 100.
4. Each time-sequentially linked protosentence after the first increases value by 100.
5. Certain constructions get bonuses of 200: the if-then-else construct and the when-X-determine-Y.
6. Any protosentence produced by multiple applications of the same aggregation rule gets a large negative value.

Figure 12. Preference rules.

One of the surprising discoveries of this work, seen in all of the cases investigated, is that the task of text generation is dominated by the need for brevity: How to avoid saying things is at least as important as how to say things. Preference Rule 1 introduces a tendency toward brevity, because most of the Aggregation rules consume two or three protosentences but produce only one, yielding a large gain in score. Sentences produced from aggregated protosentences are generally briefer than the corresponding sentences for the protosentences consumed. For example, applying Rule 1 to the pair:

> "When you permit[5] the alarm system, call the Fire Department if possible. When you permit the alarm system then evacuate."

yields,

> "When you permit the alarm system, call the Fire Department if possible, then evacuate."

Rule 3 introduces the sensitivity to advice. We expect that this sort of advice taking does not need to be elaborate — that being able to advise that a term is *good* or a term is *bad* is adequate.

Rule 6 is somewhat of a puzzle. Empirically, a sentence produced by reapplication of an Aggregation rule was always definitely unacceptable, primarily because it was awkward or confusing. We do not understand technically why this should be the case, and some say it should not be. We do know that this rule contributes significantly to overall quality.

Sentence Generator Module

The Sentence Generator (Figure 13) takes the final ordered set of protosentences produced by the Hill Climber and produces the final text, one sentence at a time. Each sentence is produced independently, using a simple context-free grammar and semantic testing rules. Because sentence generation has not been the focus of our work, this module does not represent much innovation, but merely establishes that the text formation work has been completed and does not depend on further complex processing.

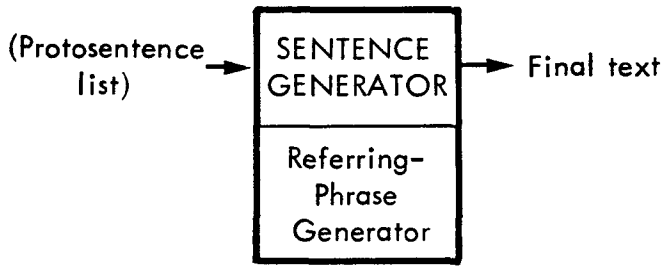

Figure 13. Sentence Generator module input and output.

The single significant innovation in the Sentence Generator is the Referring Phrase Generator, the only part in which prior sentences affect the current sentence. The Referring Phrase Generator keeps track of what objects have been referred to, and how. It presumes that objects previously referred to are in the reader's attention and that after they have been identified by the first reference, subsequent references need only distinguish the object from others in attention. This process is equivalent to the one described by [6] developed for this research. It knows how to introduce terms, refer to objects by incomplete descriptions, and introduce pronouns. However, none of our examples has exercised all of the features of Levin and Goldman's algorithm.

Output Text

Applying all of this machinery in our example, we get the result shown in Figure 14. Note the paragraph break, a product of a factoring rule (the first rule in Instructional-narrate) in the Problem Solver module.

[5] This way of using "permit" is unfamiliar to many people, but it is exactly the usage that we found in a manual of instruction for computer operators on what they should do in case of fire. In the course of attempting to produce comparable text we accepted the usage.

Whenever there is a fire, the alarm system is started, which sounds a bell and starts a timer. Ninety seconds after the timer starts, unless the alarm system is cancelled, the system calls Wells Fargo. When Wells Fargo is called, they, in turn, call the Fire Department.

When you hear the alarm bell or smell smoke, stop whatever you are doing, determine whether or not there is a fire, and decide whether to permit the alarm system or to cancel it. When you determine whether there is a fire, if there is, permit the alarm system, otherwise cancel it. When you permit the alarm system, call the Fire Department if possible, then evacuate. When you cancel the alarm system, if it is more than 90 seconds since the timer started, the system will have called Wells Fargo already, otherwise continue what you were doing.

Figure 14. Final fire-alarm text from KDS.

Conclusions and Prospects

The development of KDS highlights several aspects of the task of writing that strongly influence text quality. The overwhelming importance of brevity, seen in both the Knowledge Filter and the Preference rules, is striking. Writing is seen here as a constructive activity rather than simply as interpretive. That is, it is not so much a mapping between knowledge representations as it is the creation of new symbolic objects, not equivalent to older ones, but suitable for achieving particular effects. The image of writing as a kind of goal pursuit activity helps us to factor the task into parts. The task (and the program) is occupied with finding a good way to say things, not with establishing feasibility of saying them.

The KDS development has also identified important features of the problem of designing a knowledge-delivery program. The defects of the Partitioning paradigm are newly appreciated; the Fragment-and-Compose paradigm is much more manageable. It is easy to understand, and the creation of Aggregation rules is not difficult. The separation of Aggregation and Preference actions seems essential to the task, or at least to making the task manageable. As a kind of competence/performance separation it is also of theoretical interest. Knowledge filtering, as one kind of responsiveness of the writer to the reader, is essential to producing good text.

The importance of fragmenting is clear, and the kinds of demands placed on the Fragmenter have been clarified, but effective methods of fragmenting arbitrary knowledge sources are still not well understood.

In the future, we expect to see the Fragment-and-Compose paradigm reapplied extensively. We expect to see goal-pursuing processes applied to text organization and style selection. We expect distinct processes for aggregating fragments and selecting combinations on a preference basis. We also expect a well developed model of the reader, including inference capabilities and methods for keeping the model up to date as the text progresses. Finally, we expect a great deal of elaboration of the kinds of aggregation performed and of the kinds of considerations to which preference selection responds.

References

[1] Badler, N.I., "The Conceptual Description of Physical Activities," In *Proceedings of the 13th Annual Meeting of the Association for Computational Linguistics, AJCL Microfiche, 35,* 1975.

[2] Carbonell J.R., and A.M. Collins, "Natural Semantics in Artificial Intelligence," In *Proceedings of the Third International Joint Conference on Artificial Intelligence,* 1973, 344-351.

[3] Davey, Anthony, *Discourse Production,* Edinburgh University Press, Edinburgh, 1979.

[4] Swartout, William R., "Producing Explanations and Justifications of Expert Consulting Programs," *Technical Report TR-251,* MIT Laboratory for Computer Science, January 1981.

[5] Heidorn, George E., "Natural Language Inputs to a Simulation Programming System," *Technical Report NPS-55HD72101A,* Naval Postgraduate School, 1972.

[6] Levin, J.A., and N.M. Goldman, "Process Models of Reference in Context," *Research Report 78-72,* USC/Information Sciences Institute, 1978.

[7] McDonald, D.D., "Natural Language Production as a Process of Decision-Making Under Constraints," *PhD Thesis,* MIT, Dept. of Electrical Engineering and Computer Science, 1980.

[8] Meehan, James R., "TALE-SPIN, An Interactive Program that Writes Stories." In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977.

[9] Nida, Eugene, *Toward a Science of Translating,* E.J. Brill, Leiden, 1964.

[10] Schank, Roger C., and the Yale A.I. Project, "SAM — A Story Understander," *Research Report 43,* Yale University, Dept. of Computer Science, 1975.

[11] Simmons, R., and J. Slocum, "Generating English Discourse from Semantic Networks," *Comm. ACM 15, 10* (October 1972), 891-905.

William C. Mann is a member of the research staff of Information Sciences Institute at the University of Southern California. He received the Ph.D. degree in computer science from Carnegie-Mellon University in 1973.

James A. Moore is a member of the research staff of Information Sciences Institute at the University of Southern California. He received the Ph.D. degree in computer science from Carnegie-Mellon University in 1974.