FACTORS CONTRIBUTING TO EFFICIENCY IN NATURAL LANGUAGE GENERATION[1]

David D. McDonald, Marie W. Meteer (Vaughan), and James D. Pustejovsky*

University of Massachusetts at Amherst
*Brandeis University

COINS Technical Report 87-49

ABSTRACT

While differences in starting points, grammatical formalisms, and control structures make it intrinsically difficult to compare alternative designs for natural language generation systems, there are nevertheless many points of correspondence among them. To organize these points, we introduce an abstract reference model for the generation process. We then identify five specific factors with respect to that model that certain designs incorporate to make them more efficient computationally.

To appear in G. Kempen (ed.) **Natural Language Generation: Recent Advances in Artificial Intelligence, Psychology, and Linguistics**, Kluwer Academic Publishers, Boston/Dordrecht, in press. Originally presented at the Third International Workshop on Natural Language Generation, Nijmegen, Holland, August 20 - 23, 1986.

[1] Support for the preparation of this paper was provided in part by the Defense Advanced Research Projects Agency under contract number N00014-85-K-0017 monitored by the Office of Naval Research.

TABLE OF CONTENTS

1. Abstract 1
2. Introduction 1
3. Definitions and Concerns 1
    3.1 Situations and the amount of effort reqired to produce an utterance
4. A Reference Model of the Generation Process 3
    4.1 Canned text
    4.2 The necessity of a compositional analysis of situations
    4.3 Mapping to intermediate representational levels
    4.4 The model
5. Sources of Efficiency in Generation 6
    5.1 Precomputation
    5.2 Size of the steps
    5.3 Taking advantage of regularities in natural language
    5.4 Lazy evaluation
    5.5 Control sources
6. Illustration in Actual Systems 9
    6.1 Identifying the Relevant Portion of the Situation (RPS)
    6.2 Efficiency issues in the mapping stage
    6.3 The Reading Out Stage: problems due to architecture
7. Concluding Remarks 13
    7.1 Difficulties in defining a complexity metric
    7.2 An initial proposal
8. Appendix 15
    8.1 Babel
    8.2 Genaro/Mumble
    8.3 Kamp
    8.4 KDS
    8.5 Penman
    8.6 Proteus
    8.7 Text
9. References 23

1. Abstract

While differences in starting points, grammatical formalisms, and control structures make it intrinsically difficult to compare alternative designs for natural language generation systems, there are nevertheless many points of correspondence among them. To organize these points, we introduce an abstract reference model for the generation process. We then identify five specific factors with respect to that model that certain designs incorporate to make them more efficient computationally.

2. Introduction

Our goal in this paper is to provide a way to talk about the consequences of alternative architectures for the generation process. The particular structures that a given generator may have (i.e. its rules, categories, and lexical definitions) and the texts that it can produce will vary as its research team continues to work with it; however the architectural features of the design, including its choice of notations, control structures, and representational levels, will be relatively constant. It is these more fundamental design issues that give each generator its own character, and it is these issues, once the varying skills of the computer programmers have been factored out, which will make one generator design more efficient than another, i.e. able to solve the same problem with fewer steps. In our own research on generation, we have always had the intuition that our architectural choices have deep consequences for our program's runtime efficiency. This paper is an initial attempt to ground that intuition in a proper comparative analysis.

3. Definitions and Concerns

For the purposes of this paper we take natural language generation to be the process of deliberately producing a natural language utterance in order to meet specified communicative goals in a particular situation. We thus will not consider here systems that produce sentences randomly such as grammar checkers (e.g. Friedman, 1969), systems for language instruction (e.g. Bates, 1980), generation done during mechanical translation (e.g. Luckhardt, 1987), or any other kind of program for text production that does not act in the service of an actual synthetic speaker talking to a human user for a purpose.

"Utterances" are for our purposes fluent sequences of words of any length that would be appropriate as one turn of a conversation, e.g. anything from a single word expletive to a lecture. While we do not care whether the utterance is actually spoken (as opposed to printed), we do assume that the word sequence is accompanied by its phrase structure and other grammatical information necessary to define its intonational structure.

The generation systems that we are concerned with do not operate in isolation. They are ancillary processes in the employ of some other system: a database query system, expert diagnostician, ICAI tutor, etc. This system, which we will refer to uniformly as "the underlying program", is the source of the communicative goals; it is what is in the situation that defines the context. The conversation is between the underlying program and its human user. We take it for granted that it is sensible to talk about "the generator" as a faculty independent of this program: The two may interpenetrate each other's processing and liberally influence each other's design, but they dwell in two distinct ontological domains, are arguably distinct physically in the human brain, and are profitably kept separate in computer systems.

We recognize that many of the limitiations on the competence of today's generators--their expressive capacity to aptly use a wide range of syntactic constructions and vocabulary--stem not from the generator designs but from limitations of the underlying programs to conceptually model and represent the range and richness of situations that people can. From this perspective the

limitations of particular generators will not for the most part be a concern of this paper: We presume that the authors of the various designs believe that they will be able to extend the linguistic competence of their generators over time in step with extensions to the situational competence of the underlying programs they work with, and that the major architectural features of their designs will not have to change significantly to do so.

3.1 Situations and the amount of effort required to produce an utterance

It is ae little appreciated fact that not all utterances take the same effort for humans to produce. Consider, for example, what it feels like to greet a colleague as compared to being interviewed for a job. The greeting is "automatic" and subjectively effortless; it requires very little of our attention and can be accomplished well enough even when one is mentally impaired due to lack of sleep or inebriation. On the other hand, a job interview, a lecture on a new subject matter, or an important meeting all require our full attention to do well; we often speak more slowly in these situations and sense a greater deliberation in our choice of wording and phrasing.

We believe that the key element governing the difficulty of utterance production is the degree of familiarity of the situation. Completely familiar situations (e.g. answering the phone) require the least amount of effort; highly unfamiliar situations require a great deal more. While we are not yet prepared to present a formalization of this relationship, we do see it already as an important desideratum for generator design: a commonplace situation should require less effort for the generator to handle, i.e. fewer steps in the processing, less scratch memory consumed, less elaborate reasoning required. (See further discussion section 7.2)

What then is a "situation" such that it may be more or less familiar? Our use of the term is in the same spirit as Barwise and Perry (1983). A speaker (underlying program) is "in" a situation, i.e. located in space and time in the presence of a given audience and set of props and as the result of a specific history. These and other contextualizing factors are properties of the real world to which the speaker bears a semantic relationship, in the philosopher's customary sense. Situations are formally only partial descriptions of the world. This design provides a very useful encapsulation of effects, e.g. one is not immediately causally affected by events on the other side of the world.

The speaker maintains his own mental model of the situation he is in--an interpreted description of the actual external situation. This is an identifiable part of his mental state along with, of course, his own goals, perceptions, beliefs, etc. (part of the situation, but not a public part). It is this mental state[1] that controls what the speaker says. Following common practice in AI, we presume that the phrases produced by the generator refer to individuals in the semantic model that the program has of its situation. These individuals, "objects" denoting people, events, states of affairs, etc., constitute a "model-level" of representation.

The situation can be distinguished from the generator's "reference knowledge": its grammar, rules of usage and style, preconstructed phrases, etc. While the situation is a dynamic structure which controls what is said at a particular moment, the reference knowledge is static and is central to the control of what the generator actually does in producing the utterance. All variance in the produced text is thus by definition due to the situation. (See appendix for examples of reference knowledge in particular generation systems.)

[1] As for the form in which the underlying program models the situation that it is in, it is most useful in this paper to leave it unspecified. Given today's programming techniques, any number of different means could be used, all equally effective from an engineering point of view: the situation might be explicitly represented as statements in a theory and queried by predicates whenever a situation-dependent decision must be made; it might be given as distributed, active routines that operate directly as the situation changes; it might be completely implicit in the program's structure and execution patterns. As yet we see no basis by which to say that any one of these alternative techniques is psychologically correct and the others wrong.

Not all of the objects and parameter values that make up the underlying program's model of its situation will actually be relevant to the actions of the generator; that is, only a few of the individuals in the program's world model will actually be mentioned in any particular utterance. This has led us to posit an abstraction of the full situation, tautologically named "the relevant portion of the situation" or RPS (see examples in section 6.1). The RPS is defined to be all and only those aspects of the underlying program's model of its situation that the generator refers to in the production of a specific utterance. As an operational definition, imagine that we trace the execution of a generator and put a green mark on every underlying program entity that the generator refers to during its execution. When the process has completed the set of all green entities constitutes the RPS.

4. A Reference Model of the Generation Process

The only effective methodology for comparing different systems is to refer them all to a single model. Picking out one of the systems as the model and translating all of the others into its terms would be presumptuous, and, given the present state of the field, not very fruitful since research projects differ in which parts of the problem they have focused on. Accordingly, we have developed a new, independent model which does not, so far as we know, directly mimic any of the generator designs being persued today. We have kept our model deliberately abstract so that we can apply informatively to any generation system as a rational reconstruction.

We begin by outlining a simplification of the model to set the tone and direction of our explication; we then incrementally elaborate it in the remainder of this section. We identify the following three steps in the generation process:

1. Identifying the speaker's situation.
2. Mapping that situation onto an utterance.
3. Reading out the utterance.

According to this model, the problem that a generator solves is how to navigate from its underlying program's position in situation space to a position in utterance space, i.e. to pick out which utterance in the language is appropriate[2] to use in the particular situation that the underlying program is in. To do this the generator must first identify where in its situation space the program actually is, then apply some mapping function to that point in order to find or calculate a corresponding utterance, which is then read out.

4.1 Canned text

In the simplified form just given, the model is appropriate only for generation in the most familiar of situations, such as greetings or introductions (e.g. "Pleased to meet you"). The equivalent of this in a generator is "canned text". As an example consider the error statement (`ferror`) in the following excerpt from the code of the Mumble-86 implementation:

[2] Note that we do not assume that the utterance is the "most" appropriate one for the situation. People do not seem to need to be optimal in their choice (if they were there would be no need for revision); accordingly we see no need to demand it of our machines.

```
(defun  Realize  (message)
   (typecase  message
      (bundle  (funcall (driver (bundle-type message)) message)
      (kernel  (realize-kernel message))
      (otherwise
         (ferror "Unanticipated kind of message - ~A"
                 message))))
```

This error statement contains a typical canned text. In the Lisp programming language it is an instance of a "format statement": a string of words enclosed in quotation marks that will be read out onto the user's screen when the statement is executed. As it is being read out, special indicators within the string such as the ~A are interpreted as instructions to substitute into the string at that point the printed form of the value of some indicated Lisp expression, in this case the variable message.

Its limitations aside (see below), the execution of a "canned" text that has been incorporated directly into an underlying program's code is surely the most efficient generation process possible. In terms of our model, we see that the processor reaching and executing the ferror line of code is identical with initiating and carrying out its generation process. The first step in the sequence, "identifying the speaker's situation", requires no effort; it is a natural side-effect of the program's regular actions. The mapping step is equally trivial, since an explicit link to the utterance, i.e. supplying the word string as an argument to the generation function, was preconstructed by the programmer at the time he wrote the code. The reading out step is done by a primitive of the programming language, and is, indeed, the only substantive activity that a canned-text generation system does.

As this particular example contains a variable in the string, it is in fact not so much a canned text as a "template" into which variable information can be inserted, in this case the identity of the unknown type of message. The relevant part of the program's situation is not just being in the state of executing that line of code but also the value of the local variable "message". Such a use of variables is an prototypical illustration of a "distributed state", the norm in all consequential underlying programs and generators. Because of distributed states, we must appreciate that the distinct "positions" in an underlying program's situation space--the possible situations that it can be in--are not points but highly structured entities, and that the mapping from these situations to utterances can be quite complex.

4.2 The necessity of a compositional analysis of situations

The canned text approach is effective as the generation mechanism for simple programs such as a compiler, because the programmer can easily anticipate which elements are relevant and bring them together into a fixed mapping. But as the relationships between programs and their users become more complex, it becomes progressively more difficult to anticipate at the time the program is written what will be relevant to communicate. When programs must construct the basis from which they are going to generate they then begin to fall under a body of considerations that until recently have only been applied to human use of language. These considerations revolve around whether utterances should be viewed as the result of a fixed mapping or a compositional one.

The classic argument for compositionality of a natural language relies on the so-called "creativity" of language. The apparent fact that the number of utterances in a natural language is unbounded is one of its more widely remarked upon properties and a core tenet of modern linguistic theory. The classic argument for creativity uses the idea that one can continually add further adjuncts to sentences to establish that there can be no longest sentence and therefore no finite number of sentences (see Chomsky, 1957). Linguists argue from this observation that the total set of utterances cannot possibly be pre-formed in the mind since the mind must be taken to have a finite

manifestation. Instead, the generative linguist holds that utterances are assembled dynamically from a finite set of parts (i.e. words and grammatical relations) in accordance with a grammar.

This conventional argument for the creativity of natural language is overly strained: who has actually heard a 500 word sentence? In contrast, anyone who studies generation has available a far more reasonable and commonsense account of creativity, namely that one continually uses new utterances because one is continually faced with new situations--a conclusion that follows directly from our model. The counterbalance to creativity is the "efficiency" of language (Barwise & Perry, 1983): the fact that many utterances do reoccur countless times (e.g. "Where did you go for dinner last night?"). This also has a direct account in the fact that many of the situations one finds oneself in are similar.

The assumption of a finite mind applies to situations as well. An interpretation of the world, a situation, must consist of a finite set of relations over a finite vocabulary of elements. The unbounded size of a natural language thus becomes ultimately a consequence of the introduction of new situational terms into the speaker's state set over time as a result of learning or perception.

In a computer program, the finite vocabulary of situation-defining elements is likely to be the values of the reference variables in its code (such as "`message`" in the previous example), or the presence or absence of certain structured objects in its database (world model). In addition, an AI program will normally have semantic reference knowledge which supplies the characterizations by which the individuals in the world model are to be understood; this often takes the form of a semantic network which may play an active or a passive role in the program's operation. This is a distributed manifestation of the program's state: There is no single entity, such as the line of code presently being executed, that we could point to as "the" representation of the state and therefore could associate in any simple way with a set of actions the generator was to take.

4.3 Mapping to intermediate representational levels

Once the situation has been identified, the generator must map it to an utterance. No serious system today moves from its initial situation to an utterance in one leap. This is surely no accident: Program designers do not add structure to a system just to excercise their creativity as linguists or AI researchers. Indeed, as we discuss in section 5.2, the introduction into the generation process of intermediate abstract entities, along with the representational systems and reference knowledge to manage them, is a natural means of increasing efficiency.

Representation in generation is usually not a straightforward case of having data structures (the representation) manipulated by operators (the process) to form new data structures: Structure and operation are often combined in the same formal device and elements of the structures are often not freely examinable. Even though one talks about a structure being built or selected when some specification is mapped to it, this can often equally well be seen as a process being set in motion. To avoid taking a stand on such questions of algorithmic design in the formulation of our model, we will construe levels not in terms of structures but in terms of theoretical vocabulary. Notions like "specialization", "focus", "subject", and "consonant doubling before *+ing*" are theoretical concepts identifying linguistic phenomena or structural abstractions. Linguists and AI researchers tend to agree that these concepts can be grouped into families according to the kinds of information they reference and where they occur in theoretical descriptions and rules. We can coordinate the identification of common levels across very different generator designs by attending to what theoretical vocabulary a design's particular level makes use of without factoring in its particular algorithms and data structures.

4.4 The model

We can now give our model in its full form. Examples of its application to actual generators appears in section six and the appendix. We will uniformly refer to the intermediate abstract levels as *specifications*. We will use *mapping* as the general term for moving between levels by applying a function to individual elements at the source level to arrive at elements or specifications for elements at the target level. *Realization* will refer to the usually more elaborate and more composite activity of processing a specification to produce the thing it is a specification of; it may involve mapping. Because of the sequential and incremental nature of the generation process, what were originally given as steps might now be better characterizes as *stages*. Most designs will have some activity at all their stages and representational levels simultaneously.

IDENTIFY THE RPS The logically first stage in any generation process is to *identify those elements of the underlying program's model of its situation that are relevant to the content of the utterance and the control of the processing (the "RPS")*. This need not happen all at once but may be interleaved with other stages. As we will discuss below, the effort to "identify" the RPS will include not just the effort of determining what information will be included and what will not, but also that of bringing together the elements of a distributed state and working out the consequences of their context-sensitive combination. This aggregation is necessary since as a field we have no way of thinking about how distributed, independent entities could bring about a single, atomic event.

MAPPING The logically second, iterated stage is to *map the elements of the RPS, singly or in combination, to a specification of the utterance at some representational level*. A specification at a given level may be a single structure, possibly added to by successive mappings from the previous level, or it may be multiple structures specifying independent aspects of the utterance that will be combined during a mapping to a later level.

READING OUT The third stage is to *read out the specification.* This final mapping must produce the actual text from the last representational level constructed in stage two.

5. SOURCES OF EFFICIENCY IN GENERATION

Generator designs gain efficiency according to the directness by which they move from situations to utterances. The optimal design with no wasted actions and a direct bridge between situation space and utterance space is unlikely ever to be developed: The conceptual distance between the two spaces is too great. The best we can do is to increase the sophistication of the specifications into which we map the elements of the RPS so that fewer elements will need to be considered simultaneously in determining the mapping, and, from the other end, to look for more versatile abstractions of utterance properties so as to move utterance space closer to the terms in which situations and specifications are couched. Within this framework, we will now give five specific sources of efficiency in the architectural design of natural language generation systems. We will introduce them here, then look at specific examples from actual generation systems in section six.

5.1 Precomputation

One of the most obvious sources of efficiency in generation is precomputation. Rather than construct a specification or a mapping function from first principles each time it is needed, a system may preconstruct a parameterized structure once, as part of the system's definition, and simply apply it each time it is needed. The greater the extent to which precomputed material can be drawn on as a generator operates, the more efficient it will be.

When preconstructed parts are used, there is always a tradeoff between the amount of structure incorporated into each part and the degree to which the parts become specialized to very particular

situations. The more parts, the fewer will be required over all and the construction will take fewer steps; however, there is a danger of greatly increasing the effort required to determine which part to use.

Let us consider the consequences of the two extremes of all or no precomputation. Total precomputation gives us the equivalent of canned text: single step, situation specific, executable schemas which anticipate all of the interactions that might occur during identification, mapping, or realization. In effect, the generation process becomes a matter of instantiating and executing exceptionally sophisticated format statements. The most awkward consequence of this extreme is the astronomic number of these statements that would have to be defined and stored: many times larger than the cross product of the elements comprising the space of all possible situations. At the other extreme is a design where every structure and mapping function is assembled anew with the generation of each utterance, with no reusable parts or decision procedures. Such a design would be claiming that there was no redundancy among situations or utterances or that the effort to instantiate a precomputed schema was exorbitantly expensive; both possibilities seem unlikely.

5.2 Size of the steps

A second consideration for efficiency in the generation process is the distance that a mapping must cover in moving between successive levels. The smaller the steps to be covered, the more efficiently each mapping can be designed. Consider a hypothetical design where the derivation for utterances is compositional, yet we move from the situation to the complete utterance in one massive step. In such a design the mapping function would do all of the work: it would require every element of the situation to be an input parameter and would construct the entire utterance as output. This caricature exhibits the poorest imaginable fit between representational levels for the mapping stage since no part of the situation can be considered independently of all the others. (Some designs based on discrimination nets have this flavor (Goldman, 1975) though they do partition utterances sequentially and usually maintain separate sentence-level grammars. See Figure 8.1.)

Since one goal of an efficient design is to reduce the amount of reference knowledge that must be considered in making each decision, designs that span the distance in smaller steps, each one involving a simpler mapping function with many fewer parameters, will be more efficient. One possible countervailing factor is the additional effort necessary to maintain the "extra" levels; this may be mitigated by the fact that multi-level designs can make extensive use of precomputed structures to minimize their cost.

Another argument that favors multi-level, heterogeneous designs over single-formalism, homogeneous designs is the engineering principle that *the more narrow and specific the demands on a process, the easier it is to develop highly efficient, special case mechanisms by which to implement it*. There is a countervailing methodological argument, however, since special-purpose mechanisms do take extra time to design; consequently, initial results may be better achieved with a single-formalism design. (See McDonald, 1984, for discussion.)

5.3 Taking advantage of regularities in natural language

Natural languages are very complex, but are systematically organized. Because of this, the properties of any utterance are interdependent, with a redundancy that permits the presence of some properties to predict certain others. This is appreciated in more efficient designs in the reference knowledge that they apply during the process. For example, if the generator is realizing a two-argument transitive verb then it need only determine the surface position of one of its two arguments (e.g. determining the subject on the basis of focus). To independently look for positive criteria for the syntactic positioning of both arguments would mean wasting actions that could have been avoided if the generator had been more aware of linguistic dependencies.

While this efficiency source may seem obvious and automatic, we must point out that linguistics is not finished: it is still not clear just what the actual regularities of language form and usage are. Consequently, different linguistic theories and approaches to planning can have quite significant impacts on efficiency when they are applied to generation.

5.4 Control sources

Control in a computational system is the determination of the sequence in which a set of actions will be taken. Efficient control means that every action contributes to the goal; inefficiency comes from redundant actions, backtracking, or pursuing tangents. For a given process, its control problem can be construed as the problem of how to select the right sequence of actions from the space of all the possible sequences that the process' notation allows. In general the space will be very highly structured, since most arbitrary action sequences will not lead to sensible results. The question for designers is how this space is to be defined and how the process is to navigate within it.

There are two extremes in the design of a sequence space: implicit or explicit. With completely implicit control, the sequence is supplied from outside, directly as a list of actions, and simply executed. With completely explicit control, the space is implemented as a body of conditional tests that gate and order actions; which sequence to follow is then determined dynamically as the process runs and the tests are evaluated. In the implicit control design, control rests in the externally supplied sequence or the process that was responsible for its construction. In the explicit control design, control rests within the process itself.

Processes based on an implicit sequence space are more efficient: They expend no effort on control decisions because they have all been made by an earlier process. Of course when considering the efficiency of the system as a whole, the effort of the process that choose the sequence must be factored in--a reduction in the effort of one process is not a net gain if it increases the effort of another. Designs with explicit sequence spaces expend an appreciable amount of activity as "overhead" that does not contribute to their real goals but is rather spent determining which of the possible sequences is appropriate in the situation at hand.

5.5 Lazy evaluation

Another source of efficiency is a processing technique that has been called *lazy evaluation.* By this we mean delaying the evaluation of an expression until the point in the process when it is actually going to be used. Efficiency is gained through a thoughtful ordering of the steps of a process so that information is not requested before it is available. An obvious example is delaying the decision of whether to use a pronoun until the point of the reference has been reached in the linear sequence of the utterance and all of the left context is known (see example in section 6.3).

In extreme cases, a poor ordering of computations not only can increase the amount of work that is necessary, but cause backtracking as well. Consider the following sequence of actions:

1. compute the number of the verb
2. place the arguments
3. compute the number of the nouns

Ordering step (3) after step (1) creates a redundancy since the number of the subject must be determined before the number of the verb can be computed, then determined again to mark the noun. However, the more serious flaw in this sequence is the ordering of steps (1) and (2). Until the arguments are placed, you cannot be sure which will be the grammatical subject. The wrong choice would necessitate backtracking to recompute the number of the verb.

A complementary aspect of lazy evaluation is employing intermediate representations that retain the results of any computations that might be useful later, so that the calculation will only need to be

done once; for example maintaining an explicit surface structure representation to facilitate subject-verb agreement. A possible countervailing factor to explicitly retaining early results is the cost of the representation. Unless the timing between steps is so exact that results can be passed implicitly through the equivalent of functional application, the cost of assembling and maintaining the representation may exceed the cost of recalculation. It will consequently be easier to take advantage of this source of efficiency if a design already uses a series of intermediate representations as part of its normal effort.

6. ILLUSTRATION IN ACTUAL SYSTEMS

In the previous section we enumerated several sources of efficiency, design options that enable a system to take fewer steps to solve the same problem. In this section we use the reference model presented in section four to organize specific examples of these sources. We begin by contrasting how different systems *identify the RPS*, specifically whether it is handed to them by their underlying program or they must identify it through their own effort. We then consider efficiency issues in the *mapping* stage, specifically control, taking advantage of regularies, and precomputation. Finally, we consider how certain control designs in the *reading out* stage can effect a system's intrinsic competence.

6.1 Identifying the Relevant Portion of the Situation (RPS)

According to our reference model, the first stage in generation is to identify the portion of the underlying program's model of its situation that is to be incorporated into the utterance or used for process control. Identification of the RPS may be automatic, where it is done for the generator by the underlying program as part of its input, or it may instead be the generator's active responsibility and the first action that it takes. In the abstract, which of these is preferable is straightforward: It is more efficient to let the underlying program be responsible for identifying what is relevant to the utterance. This is especially true in a process-based underlying program where the identification is just a side effect of the program's execution (see section 5.4).

It may happen that the program does not have a sufficiently rich model of its situation to be capable of providing the information necessary for generation. For example, in order to generate cohesive text, information about the coherence relations among individual propositions may have to be supplied. If the underlying program is simply a database query system, the generator itself may have to supply a model of possible coherence relations.

Two systems that differ in which component is responsible for identifying the RPS are PROTEUS (Davey, 1974) and TEXT (McKeown, 1985). In Proteus, which played tic-tac-toe and produced paragraph length descriptions of the games (see Figure 8.6), the RPS was the sequence of moves in the game, annotated according to their role, e.g. "threat" or "fork". Since the underlying program was process-based, this information was available as a side effect of its actions, and could be given to the generator without any extra effort. The coherence was provided by virtue of the underlying coherence of the actions themselves.

In McKeown's TEXT program, which produced paragraph length definitions requested by a user (see Figure 8.7), the underlying program was a conventional data base. It would be difficult (perhaps impossible) for that underlying program to identify the RPS, since it has no basis for coherence. TEXT used the user's request to filter the knowledge base and determine the relevant information, and it used preconstructed schemas to organize the information and provide coherence relations. Preconstructing schemas for definitional paragraphs allows the generator to impose coherence relations more efficiently (see section 5.1).

6.2 Efficiency issues in the Mapping Stage

In our model, a mapping is the constructive relation between two successive representational levels. Comparing mapping techniques across designs can consequently be a complex business, since except for the first and last levels, the RPS and the text, designers are free to choose whatever levels best fit their notions of how generation is done. Furthermore what is a "level" and what is a "mapping" can be a matter of judgement: The quite common use of data-directed control in generation systems blurs the lines between static representations and active transition processes.

Our working definition for level and mapping is quite pragmatic. A representational level is a set of expressions *assembled specifically for the utterance being generated.* All the expressions at a given level will have been constructed from a common vocabulary of terms and connectives; the nature of the vocabulary will establish whether the level is semantic, syntactic, logical form, etc. A mapping is a process that draws on information at one representational level (a "higher" one), plus some fixed body of reference knowledge (e.g. the code of the process, or some set of tables), to construct or add to a second ("lower") level.

The primary efficiency issues that will concern us for the mapping stage involve regularities (Section 5.3), sources of control (Section 5.5), and the possibilities for precomputation (Section 5.1). Mappings will tend to take one of two general forms: Either (1) the mapping will be controlled by its reference knowledge (taking the higher level as a parameter), or (2) it will be controlled by the higher level (drawing on its reference knowledge as needed). The Appendix provides a set of diagrams that show the representational levels and control regimenes used in mapping between them for seven different generators from the literature.

Control is given to the higher level when there are comparable structural regularities between the levels and the work that the mapping must do is simply looking up correspondences. Usually such mappings are implemented as data-directed processes with the expressions at the higher level interpreted literally as mapping actions to be carried out. (In the Appendix diagrams, this type of mapping is given as a downward pointing arrow.) These designs are a case of the action sequence (i.e. the steps of the mapping) being supplied by an earlier process, the one that constructed the higher level. As discussed in Section 5.4 we claim that this makes them the most efficient design for moving between levels because the action sequence has already been tailored specificially to the case at hand and consequently no effort needs to be expended on control decisions.

When there is nothing at the higher level serving as the basis for constructing the lower one, then that basis must be supplied by knowledge embedded in the mapping process. In the cases we have examined, this knowledge takes the form of some general model of the space of the possible structures which the lower level can have. Carrying out the mapping involves using the model to query the higher level, thereby determining what particular lower structures should be built. (In the Appendix diagrams these mappings are given as upward pointing arrows plus braces.) In the terms of Section 5.4, this is a case of control based on an explicit sequence space, since the model must define all of the possible mappings that might occur and then test the higher level to determine which one is appropriate. These tests are control decisions determining the eventual set of actions that will actually construct (i.e. "map to") the lower level. The additional effort expended on these control decisions makes such designs relatively less efficient.

We will illustrate the alternatives just sketched with the mapping that takes a generator from a level where the information is encoded in a nonlinguistic form, often as propositions expressed in a predicate logic, to a level representing the linguistic relations that define the surface phrase structure and grammatical relations of final text. This mapping is part of nearly every generator we have looked at, and has been approached in quite different and illustrative ways; we will compare the two that have struck us as being the most different: PENMAN (Mann, 1983) and MUMBLE (McDonald 1984).

In the PENMAN system, a systemic grammar known as NIGEL carries out the mapping between the propositions of the input *demand expression* and the feature-based specification of the linguistic

relations that are to realize them. NIGEL represents a text by a set of abstract features that collectively specify the form and grammatical relations of constituents. The dependency relationships between features is encoded by a set of networks which organize them into disjoint sets (*systems*). The networks indicate by the connections between systems when the inclusion of a feature from one system forces the inclusion of some feature from another, linked system. The set of all pathways through NIGEL (connections between systems) defines the set of all possible feature combinations and thereby all possible natural language texts.[3]

For PENMAN, the task of the mapping is to assemble a lower level text specification (i.e. a feature set) that will adequately convey the information in the higher level demand expression. This amounts to selecting a path through the networks of the grammar; in NIGEL this is done system by system following the chain of dependencies indicated by the links between them. In a properly designed systemic grammar such as NIGEL, this means that each system is considered only once since all of the linguistic criteria bearing on it will have already been determined by selections made earlier in the systems leading into it.

Viewed from this perspective it is easy to see that NIGEL's mapping is based on an explicit sequence space; the reference knowledge--the systemic grammar--is in control. The control decisions are the feature choices made at each system. In PENMAN these decisions are carried out by *choosers*, specialist procedures that consult the demand expression and the underlying model in order to make their choice.

In contrast, the comparable mapping in MUMBLE is from the elements of a *message* to representations of linguistic phrases rather than features. Messages, technically referred to as *realization specifications*, are broadly comparable to PENMAN demand expressions though they have a more specialized organization and are taken to have been deliberately planned. The mapping is carried out by directly executing the message as though it were a program in a very special programming language (i.e. it is passed through a special interpreter). A table of correspondences is consulted, element by element, and the construction actions indicated by the correspondences are carried out. This makes the design an implicit sequence space where control rests in the message rather than the reference knowledge of the mapping--an intrinsically more efficient design.

MUMBLE's phrases can be viewed as predefined packages of features. By taking the packages as wholes, MUMBLE's mapping is spared the effort of testing (or even representing) whether those particular features can cooccur; the packaging is a given of MUMBLE's reference grammar. NIGEL on the other hand has no representation of possible linguistic form that is independent of function (i.e. the selections made by the choosers) since its phrasal specifications are only implicit in the paths through the systems and these are determined only for specific cases.

MUMBLE's preconstruction of the linguistic form is more efficient only if it does not lead to greater effort in establishing the correspondences from message elements. If a phrase cannot be selected without first making extensive tests on the message, then the net total of tests may turn out to be quite comparable with that required to individually determine each of the features that the phrase consists of. Our intuition (speaking as MUMBLE's developers) is that there is a net savings in tests, because we believe there are regularities at the propositional/message level that closely match the information packaging that linguistic phrases embody.

Systemic grammarians' prime motivation for carrying out their analyses in terms of subphrasal features is that they see the factors that go into text form as being of very different kinds (parallel

---

[3] Being a sentence grammar, Nigel only represents all possible sentences, not all possible texts generally. For texts larger than single sentences the Penman project expects to use a different organizing scheme based on a descriptive formalism called Rhetorical Structure Theory (Mann, 1984).

N.b. "paths" through a systemic grammar consist of multiple rather than just single threads because of the presence of conjunctive as well as disjunctive feature systems. This is an opportunity for a multiprocessing implementation since multiple threads can be explored without interfering with each other.

paths through the grammar), that are reconciled as a group by a "realization" algorithm once they have all been determined. We would argue that in a design like MUMBLE the same effects can be achieved provided one is careful about how much information phrases are stipulated to contain and how phrases are allowed to combine. This allows us to retain the savings implied by precomputing sets of text properties and deal with them only as units.

Briefly our design is as follows. A propositional unit at MUMBLE's message level is not mapped to a complete surface phrase (e.g. a text string) but to a constraint expression that specifies the phrase's head and the values of the thematic relations that are to accompany it. The constraint expression is then fleshed out: the order of the thematic elements fixed, temporal and other situational anchors inserted, any additional modifiers, adjuncts, or hedging verbs added. Only then will the "phrase" have its final content and be ready to be read out. MUMBLE does this with a combination of two devices: one maps the constraint expression to a set of alternative thematic orderings (and other transformational variations) from which a selection is then made; the other adds in anchors and modifiers (which originated as independent elements of the message) at points within the phrase as permitted by its grammatical structure (see McDonald & Pustejovsky, 1985, for discussion). This strategy of breaking down linguistic forms into their smallest units and allowing a versatile set of insertion and adjunction mechanisms for incorporating further minimal units is the key to facilitating a multi-factor mapping using precomputed phrases.

Using minimal units as MUMBLE does imposes as defacto order of importance on the influences on the utterance, since they are effectively considered only one at a time and the indelible selections remove alternatives from later decisions. In contrast, an approach which breaks the mapping down to the selection of individual features, as NIGEL does, and waits to realize them as sequential text only once they all have all been choosen allows an equal consideration of all of the influences on the utterance before committing to any part of its form. This is difficult to achieve in a mapping design that uses direct links to precomputed units. Laurance Danlos (1984) discusses this issue. She employs choice sets ("discourse grammars") very much like those in MUMBLE, i.e. selections between entire surface phrases, and argues persuasively that in order to make equitable, balanced decisions one has to have a large number of alternatives in a set and use very large, composite phrases. Ultimately it will be an empirical question whether the texts produced with ordered influences are good enough (e.g. people would not do better in comparable situations) or whether the architecture must be changed.

6.3 The Reading Out Stage: problems due to architecture

The point that we wish to make about the final stage of the generation process--reading out the words from the last abstract representational level and thereby producting the utterance--is not so much one of relative efficiency as of basic competence. A program may generate in a very small number of steps, but if it is incapable of ever producing certain common constructions or communicating certain kinds of information than we do not want to say that it is more efficient than other programs that can do those things but take longer as a result. We believe that certain designs have a limited competence--specifically a restriction on their ability to properly select pronouns--because of their choice of representation and control structure rather than their particular choice of rules. This makes the problem a matter of architectural decisions in the design of their generator and thus much more serious than just a weakness in an analysis. Potential architectural problems like this may be seen at all stages in different generator designs; the one we will discuss is just easier to describe.

Generators organize their syntactic stages as a recursive descent--top down--through the phrases of their sentences: main clauses are formed and organized before subordinate clauses; the verb and thematic relations of a clause before its noun phrases; head nouns and adjectives before relative clauses. As the work at this stage usually results in fixing the order of the words in the utterance, it is often interleaved with the reading out stage in a very tight coupling: When the recursion reaches a phrasal level with words at its leaves they are collected for reading out.

We can distinguish two architectural alternatives in the design of this recursive descent: One does all of the lower levels in parallel; the other does them sequentially in left to right order. The left to right order is the one taken by ATN designs (e.g. Simmons & Slocum, 1972), and by MUMBLE. The parallel order is taken by Derr & McKeown (1984), who use a Definite Clause Grammar, and also appears to be what the systemic grammar designs of PROTEUS and PENMAN do, though we do not feel absolutely confident of this on the basis of the references we have available.

Recursion in parallel is arguably simpler; certainly it is natural when a DCG is implemented in Prolog. However, it has a cost in what this architecture will allow to be represented: Parallel recursive processes in uncomplicated DCGs deliberately ignore their surrounding context, i.e. they do not carry down with them any record of what phrases were to their left or right. Unfortunately, awareness of this context is essential for the required intrasentential pronominalization that occurs when a reference is c-comanded by its antecedent (e.g. "*Floyd wanted Roscoe to get his fishing pole*"--note that this pronominalization is forced even though it creates an ambiguity). Without the capacity to represent the necessary relationships, the design has no way to do this kind of pronominalization--an intrinsic limitation in its competence. The designs that keep an explicit surface structure have a natural means of recording the information that c-comand needs, and thus have no architectural limitation standing in the way of being competent to do this type of pronominalization.

7. CONCLUDING REMARKS

When we first began this work, it was with the hope of arriving at a proper complexity measure for generation, something analogous to saying that Earley's algorithm has a worst-case complexity of G^2n^3, a formula parameterized by the size of the grammar being used and the length of the sentence being parsed. But the formulation of a complexity metric makes demands which generation, given its present state of the art, does not appear to be able to meet.

7.1 Difficulties in defining a complexity metric

To investigate the complexity of an algorithm, one must have a clear definition of what problem is to be solved, stated in terms of a relevant set of variables, and one must have a statement of the properties of an adequate solution. We know of course that the solution in generation must be some grammatical text, but that is about as useful as knowing that the solution to some arithmetic problem is the number 3--an infinite number of problems could have that solution. At the other end we are in a still worse position, since as we all know there is no clear model in the field of what the generation process actually starts from: The neighborhood gossip who tells us that "John loves Mary" surely has more on his mind that just the proposition *loves(John, Mary)*.

We cannot declare arbitrarily that the generation process begins at some well defined point, for example, the semantic "message" that many projects have pragmatically chosen as the earliest level they will work from in their research. But while the message level might be precise enough for the definition of a metric, it lacks other essential qualities. First, no two projects' message levels contain really comparable information. (Contrast for example McDonald & Pustejovsky, 1985 with Sondheimer & Nebel, 1986). Second, such a metric would beg the question since it makes no allowance for the effort required to construct the message expression, which will vary widely depending on the assumptions of the design.

In this respect, understanding systems have no adequate complexity metric either, since while the subproblem of parsing a stream of words into a grammar-defined structural description is precise enough for metrics to be defined, the question of what happens after that, of how this structural description comes to have any impact on the hearer's future behavior, is as ill-defined as the early stages of generation.

We also have no empirical or even consensus notion of the full problem that speakers are solving when they produce an utterance. It is clear that the problem is not just to convey "an idea" from the speaker's mind to the hearer's; that does not even begin to cover the reasons why people talk (e.g. for amusement, to convey sympathy, for group identification, etc.) It is also unlikely that the more recent view of generation, i.e. as another of the speaker's mechanisms for achieving his goals, is going to yield an interesting metric; talking about "goals" and "speech acts" is just putting a more interesting name to the problem, not identifying it.

7.2 An initial proposal

While we are not yet prepared to propose an entire complexity metric, we do believe we have identified one of its primary parameters, namely the familiarity of the situation. We also believe we can characterize the architectures that will turn out to rate most highly on this metric, namely those that draw on the efficiency sources that we have identified, particulary precomputation. We summarize our reasoning below.

Introspection and gedanken-experiments suggest that the simplest utterances to generate are those that are overlearned: conventional greetings, rehearsed speeches, idioms and highly stylized phrases. The only work that must go on to produce them is (1) assessment of the situation to identify it as one where the memorized utterance is appropriate, (2) retrieval of the utterance (presumably by table look up from the identity of the situation), and (3) uttering (reading out) the words.

We assume that any complexity metric will factor out the length of the recalled word string: if the memorization or overlearning is effective then it will be just as simple (i.e. minimally complex) to produce a long memorized text as a short one. Similarly at higher levels in the generation process we would expect that the length of any precomputed specification or mapping function would also be discounted as trivial in cost when compared with the effort to assemble the structure from primitives by reference to non-specific reference knowledge. It is no doubt true that the physical details of the actual human generation process must put some limit on the size and character of what can be precomputed, stored, and instantiated, but we presume that these limits are liberal enough to have no practical impact on our claims.

A more significant parameter of the metric is the familiarity of the situation (see section 3.1). Increased familiarity makes for more certain and probably easier identification, and the recognition of regularity--the reification of a situation type--makes it possible to directly associate it with memorized phrases or specifications, i.e. to precompute elements of the generation process that situation type will initiate.

The vast human capacity for categorized recognition and recall suggests to us that the space-time trade-offs for mental computation weigh heavily in favor of using space to save time[4]. Consequently, we claim that it is always more efficient to implement the mapping stage of generation as the selection and instantiation of a preconstructed schema or specification rather than assemble such structures dynamically from primitive elements each time they are used.

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.

[4] This may of course not be the case for present day computers if we want them to have a high degree of competence and still function in real time. But we still suggest that machine designs at least emulate a space-intensive design, since in the engineering of artificial counterparts of natural phenomena like language, cognitive theories are our best guidelines for achieving an extendible and robust system.

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.

In summary, we see the total effort required to produce an utterance varying in proportion with the degree to which the speaker is able to identify the situation as a familiar one and can thereby model the RPS in terms of known situational types. Since the generation process is more efficient to the extent that it can be done using preconstructed elements, greater familiarity will result in quicker generation.

8. APPENDIX

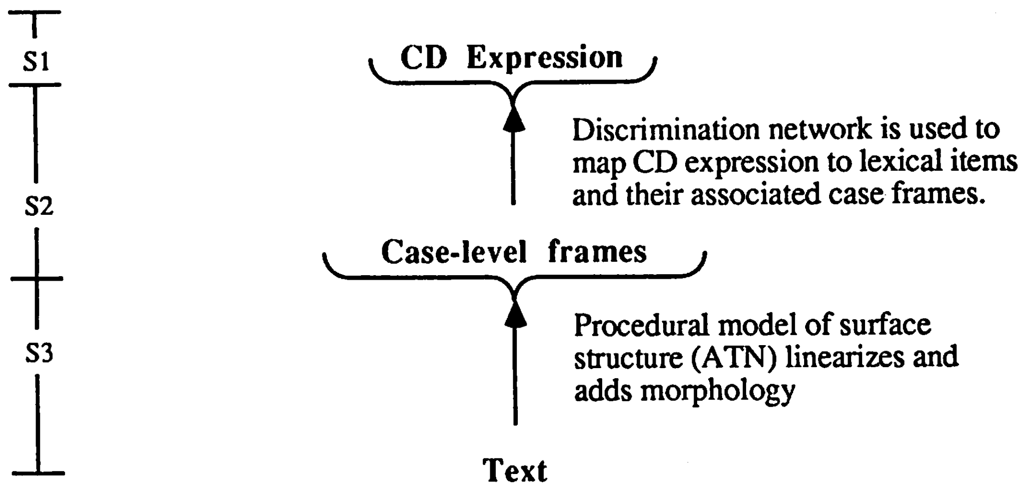
One of our major goals in this paper has been to find a way to compare different approaches to generation. In the following diagrams we attempt to capture in a uniform way how different systems compare along the dimensions we have raised in this paper: how they relate to our three stage model, what the various representational levels are, what controls the mapping between representational levels, and what the reference knowledge is. We also give a short biographical sketch of each system. Appearances to the contrary, the diagrams should not be taken to imply a sequential processing. In all these systems multiple levels are typically active simultaneously and processes may operate recursively.

We have discussed generation abstractly in terms of a series of mappings between representational levels. Our model divides those mappings into three stages: The first is characterized by considerations of what is relevant in the situation the underlying program is in (what we generate must meet the goals of the underlying program) and the third is constrained to be English or some other natural language (what we generate must be understandable to people). The second stage spans the distance between the situation space and the utterance space. (Stages are indicated on the left of each diagram.)

The representational levels are shown in bold. The mappings between them are indicated by the arrows, with descriptions of the processes on their right. We distinguish between two types of mappings based on whether the representational level is in control or the reference knowledge is in control. The down arrow indicates a mapping in which the higher representation is controlling in the mapping to the lower representation. (In some cases it is actually an executable representation, for example Mumble's surface structure.) These processes access the reference knowledge, but control decisions are not based on it. The bracket and up arrow indicate a mapping process governed by the reference knowledge. Such a process accesses the higher representation in the production of the lower representation, but control decisions are based on the reference knowledge.

Our choice of which systems to include was based on our familiarity with them and the availability of reference materials. We present them in alphabetical order by system name, to avoid any unintended implications.

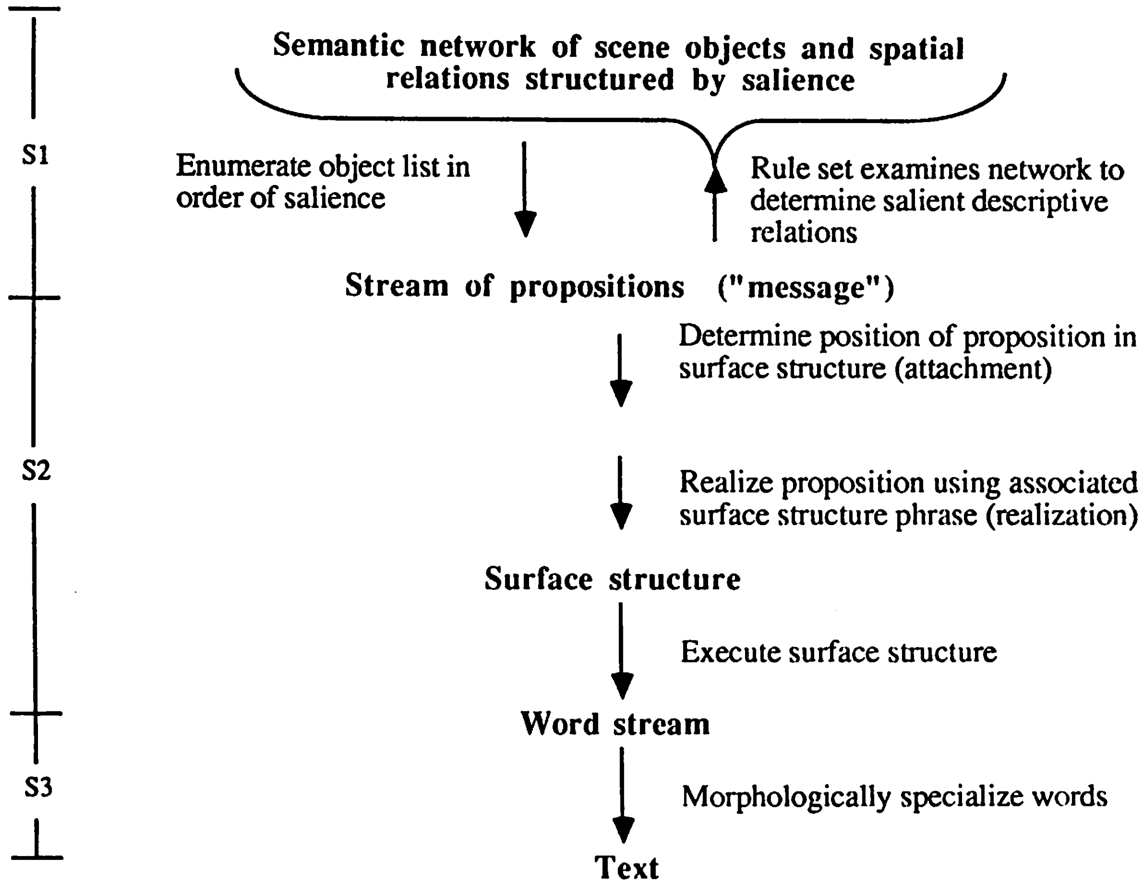
Stages:



REFERENCE KNOWLEDGE: Discrimination networks
ATN grammar

Babel (Goldman, 1975) produced sentence length text exemplifying various paraphrases of isolated Conceptual Dependency expressions. The focus of the project was on using a discrimination network for lexical choice. It has one of the fewest number of representational levels of any system in the literature, with the bulk of the linguistic work done by its ATN generator, which was developed by Simmons & Slocum (1972).

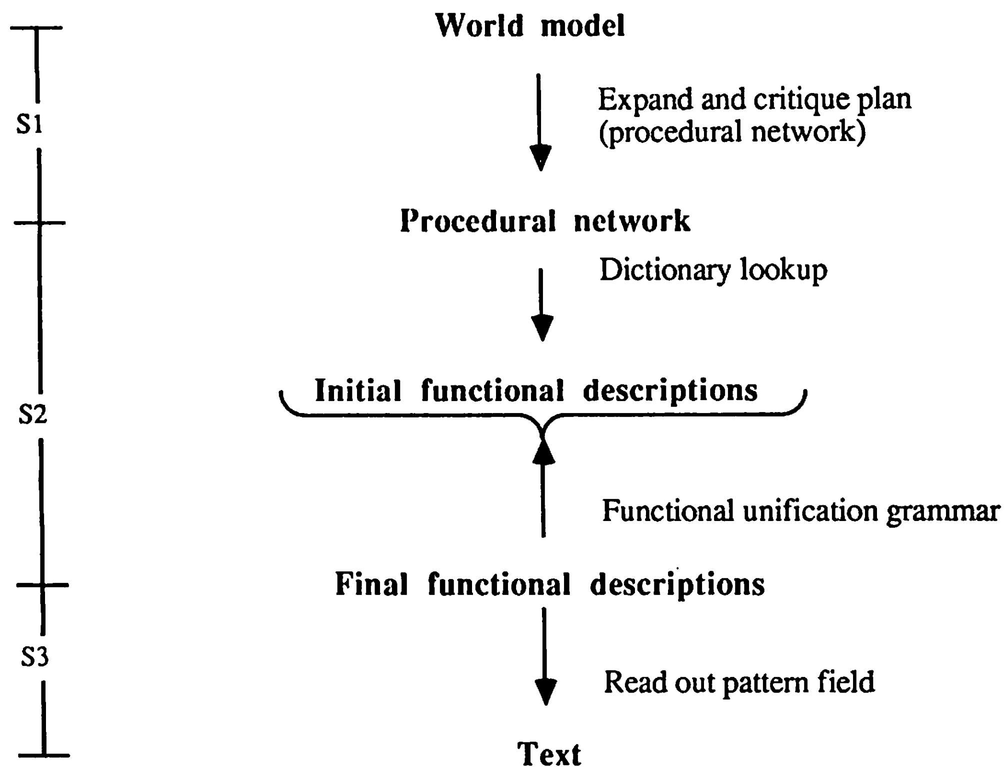
Figure 8.1 BABEL



REFERENCE KNOWLEDGE:	Genaro:	Rule set of descriptive types
	Mumble:	Set of attachment classes Set of realization classes Tables associating message elements with attachment and realization classes

This diagram represents two systems: the text planner Genaro (Conklin, 1983) and the linguistic component Mumble (McDonald, 1984). Genaro was the first substantitive text planner used with Mumble, which has been evolving since 1976. Genaro planned paragraph length descriptions of pictures of houses, using visual salience for organization. Mumble is one of the few systems that doesn't assume an intermediate representation aggregated into sentence sized chunks before phrase structure decisions are made. Mumble's attachment process allows a unit to be incorprorated into an already realized phrase structure or to become a separate sentence depending on its relation to previous text and stylistic considerations.

Figure 8.2 GENARO/MUMBLE

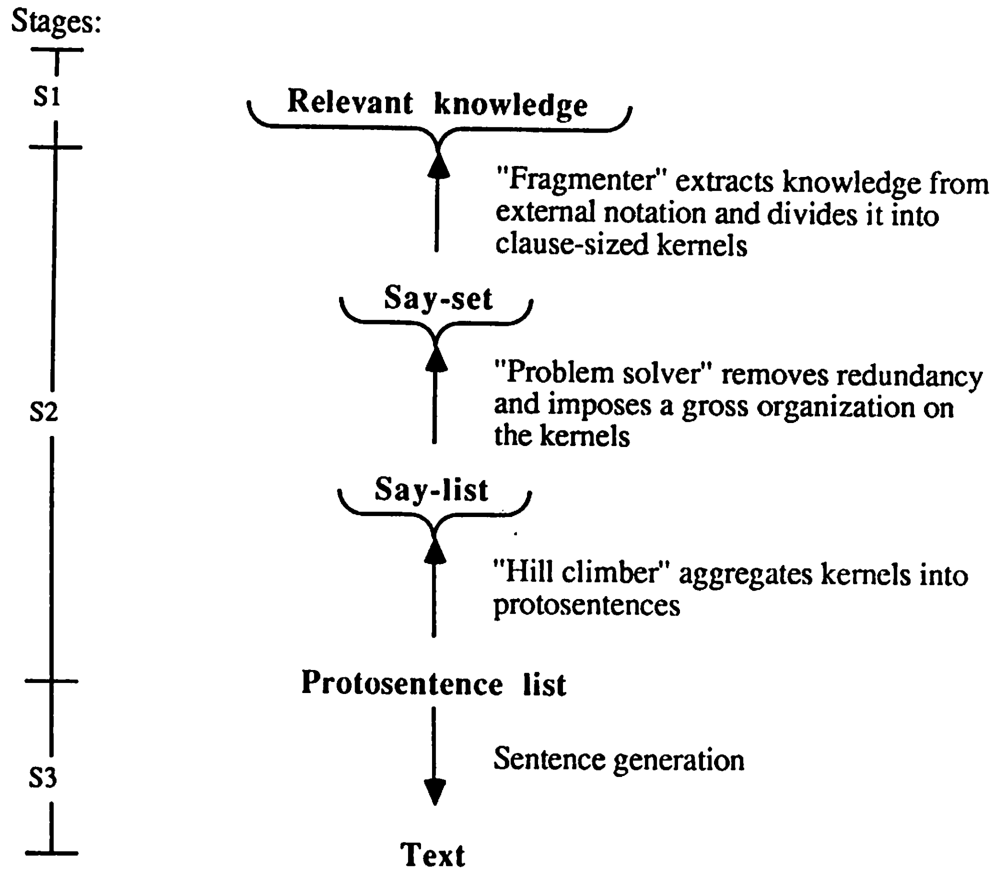


REFERENCE KNOWLEDGE:

- Plan step expansion specialists
- Plan critics
- Table associating primitive plan step and initial functional descriptions
- Functional unification grammar

Kamp (described in Appelt, 1985) did first principles planning of an utterance in a cooperative dialog. Appelt's major concern was that the axiomatization of the text planning process be on a firm foundation. This careful reasoning about what should be included in the utterance and how the information was related allowed the system to produce complex sentences.

Figure 8.3 KAMP

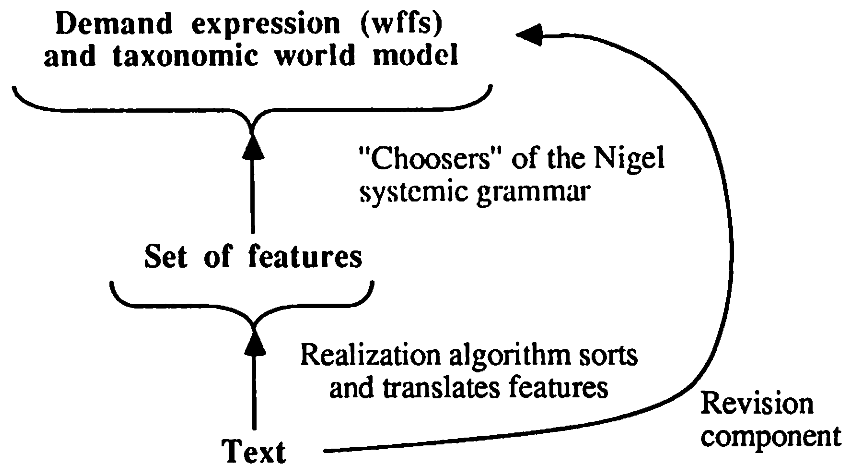


REFERENCE KNOWLEDGE: Sentence generator
Hill climbing algorithm
Problem Solver
Fragmenter

KDS (Mann & Moore,1981) produced multiparagraph text describing procedures (such as emergency fire procedures). They took as assumptions (1) a lack of isomorphism between the size and organization of objects in the underlying program and the phrases in the surface structure text and (2) a restriction that the input to the generation component must be in sentence sized chunks (contrast Mumble, figure 8.2); they then looked at how the system could produce complex sentences which were cohesive with the surrounding text.

Figure 8.4 KDS

Stages:

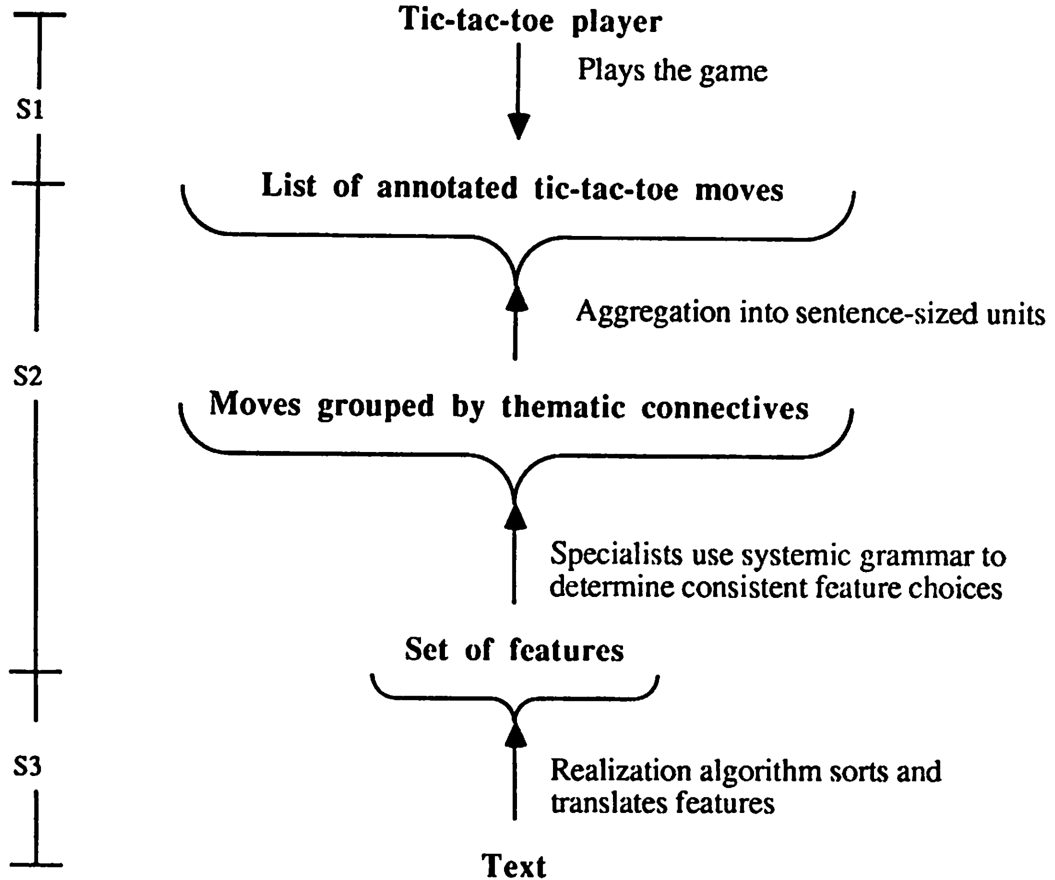


REFERENCE KNOWLEDGE: Taxonomic world model
Systemic grammar and choosers
Realization algorithm

Penman (Mann, 1983) focuses on the use of systemic grammar in generation. Nigel, Penman's grammar, (Mann & Matthiessen, 1985) is the largest systemic grammar and possibly the largest machine grammar of any kind. It produces isolated sentences using a complex system of choosers which query the demand expression for the utterance and the hierarchical world model. The other components of Penman, the planner which determines the demand expression and the revision component, are still in the design stages.

Figure 8.5 PENMAN

Stages:



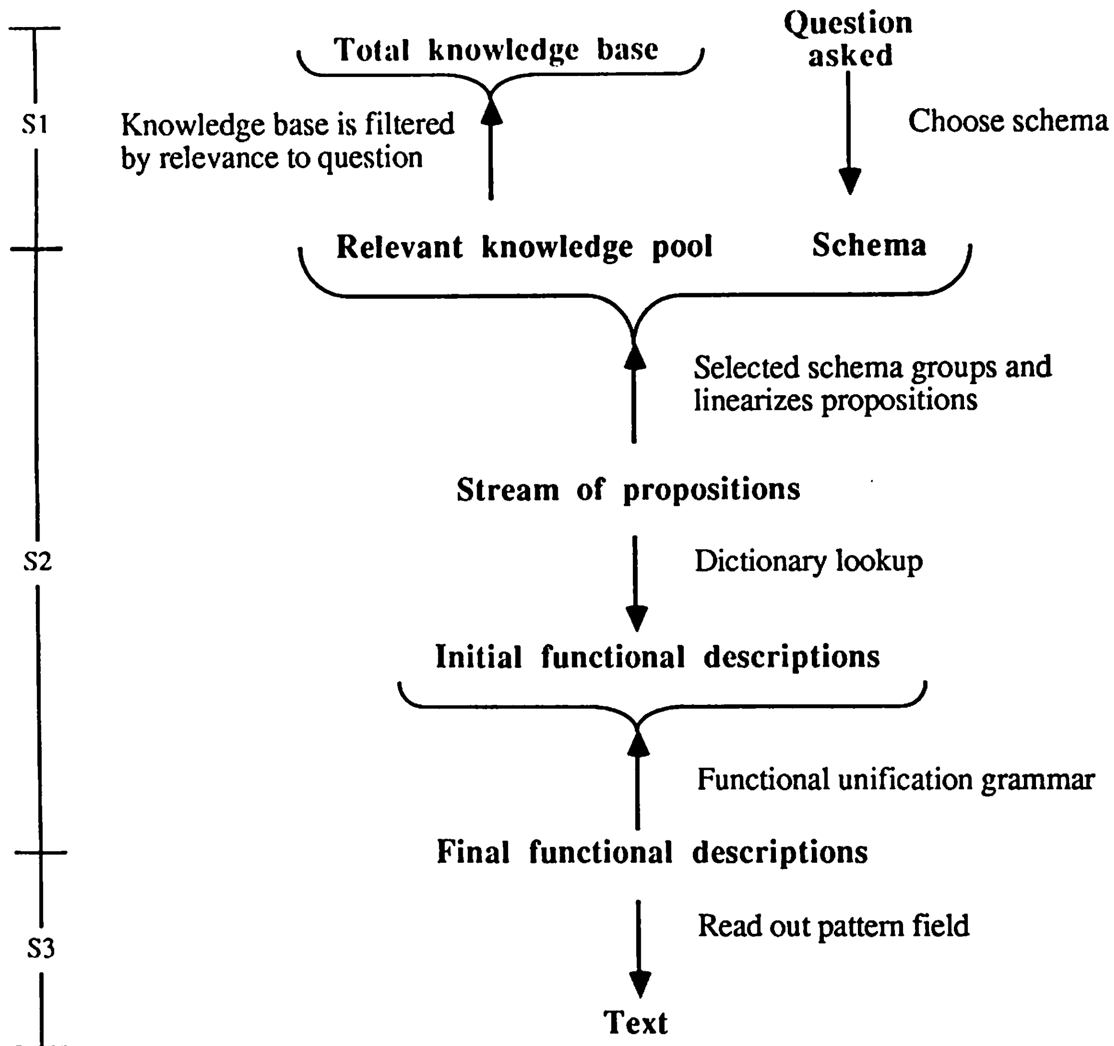
REFERENCE KNOWLEDGE:

Procedural encoding of all possible aggregations
Procedural encoding of system grammar
Realization algorithm

Proteus (Davey, 1974) is one of the earliest serious generation systems, yet the text it produced is still among the best machines have generated. Proteus both played tic-tac-toe and generated paragraph length commentaries on the games, using a systemic grammar. Since the system was both the underlying program and the generator, it was able to take advantage of the structure of its model of tic-tac-toe to organize the structure of the text.

Figure 8.6 PROTEUS

Stages:



REFERENCE KNOWLEDGE:

Predefined schemas
Association table of propositions to initial functional descriptions
Functional unification grammar

Text (McKeown, 1985) focuses on the rhetorical structure of text. It uses predefined schemas and focus information to organize definitional paragraphs in response to questions about the meaning of terms in a conventional data base. In the version of TEXT presently used at the University of Pennsylvania, the functional unification grammar has been replaced with the generator Mumble.

Figure 8.7 TEXT

9. REFERENCES

Appelt D. (1985) *Planning English Sentences*, Cambridge University Press, Cambridge.

Barwise, J. & J. Perry (1983) *Situations and Attitudes*, MIT Press, Cambridge, Massachusetts.

Bates, M. & R. Ingria (1981) "Controlled transformational sentence generation", Proceedings of the 19th Annual Meeting of the Association for Computational Linguistics, Stanford.

Chomsky, N. (1957) *Syntactic Structures*, Mouton & Co., The Hague, The Netherlands.

Conklin, E. (1983) *Data-driven Indelible Planning of Discouse Generation Using Salience*, Ph.D. thesis, Univ. of Massachusetts, Dept. of Computer Science.

Danlos L. (1984) "Conceptual and Linguistic Decisions in Generation," Proceedings of COLING-84, Stanford University, July 2-6, 1984, 501-504.

Davey A. (1974) *Discourse Production*, Ph.D. thesis, Edinburgh University; published in 1978 by Edinburgh University Press.

Derr, M. & K. McKeown (1984) "Using Focus to Generate Complex and Simple Sentences", proc. COLING-84, Stanford University, July 2-6, 1984, 319-325.

Friedman, J. (1969) "Directed random generation of sentences", CACM 12(6), 40-46.

Goldman N. (1975) "Conceptual Generation", in Schank, R. *Conceptual Information Processing*. North-Holland/Elsevier, 289-372.

Luckhardt, H.D. (1987) "Generation of Sentences from a Syntactic Deep Structure with a Semantic Component", McDonald & Bolc (eds.) *Papers in Language Generation*, Springer-Verlag, New York, in press.

Mann W. (1983) *An Overview of the Penman Text Generation System*, USC/ISI Technical Report RR-83-114.

_____ (1984) *Discourse Structures for Text Generation*, USC/ISI Technical Report RR-84-127.

_____ & Matthiessen (1985) "Nigel: a Systemic Grammar for Text Generation", in Freedle (ed.) *Systemic Perspectives on Discourse: Selected Theoretical Papers of the 9th Intl. Systemic Workshop*, Ablex.

_____ & J. Moore (1981) "Computer generation of multi-paragraph English text", JACL 7(1), 17-29.

McDonald D. (1984) "Description Directed Control: Its implications for natural language generation", in Cercone (ed), *Computational Linguistics*, Plenum Press, pgs. 403-424; reprinted in B. Grosz, K. Spark Jones, & B. Webber (eds) *Readings in Natural Language Processing*, Morgan Kaufman Publishers, California, 1986, 519-538.

_____ & Pustejovsky (1985) "TAGs as a Grammatical Formalism for Generation", proc ACL-85, Chicago, 94-103.

McKeown K. (1985) *Text Generation*, Cambridge University Press.

Sondhiemer, Norm & Bernhard Nebel (1986) "A Logical-form and Knowledge-base Design for Natural Language Generation", proc. AAAI-86, Philadelphia, August 11-15, 1986, 612-618.

Simmons R. & J. Slocum (1972) "Generating English discourse from semantic networks", CACM 15(10), 891-905.