Natural Language Generation from Plans

Chris Mellish[1] and Roger Evans

School of Cognitive and Computing Sciences
University of Sussex
Falmer

Brighton BN1 9QN
United Kingdom

This paper addresses the problem of designing a system that accepts a plan structure of the sort generated by AI planning programs and produces natural language text explaining how to execute the plan. We describe a system that generates text from plans produced by the NONLIN planner (Tate 1976).

The results of our system are promising, but the texts still lack much of the smoothness of human-generated text. This is partly because, although the domain of plans seems *a priori* to provide rich structure that a natural language generator can use, in practice a plan that is generated without the production of explanations in mind rarely contains the kinds of information that would yield an interesting natural language account. For instance, the hierarchical organization assigned to a plan is liable to reflect more a programmer's approach to generating a class of plans efficiently than the way that a human would naturally "chunk" the relevant actions. Such problems are, of course, similar to those that Swartout (1983) encountered with expert systems. In addition, AI planners have a restricted view of the world that is hard to match up with the normal semantics of natural language expressions. Thus constructs that are primitive to the planner may be only clumsily or misleadingly expressed in natural language, and the range of possible natural language constructs may be artificially limited by the shallowness of the planner's representations.

1 Introduction

Planning is a central concept in Artificial Intelligence, and the state of the art in planning systems allows quite complex plans to be produced with very little human guidance. If these plans are to be for human consumption, they must be explained in a way that is comprehensible to a human being. There is thus a practical reason for considering ways of generating natural language from plans. There are also theoretical reasons why plans are a good domain for studying natural language generation. Although there may be a great deal of material in a given plan, there is a kind of consensus among planning researchers on what sort of information a plan is likely to contain. Thus it is possible that interesting general principles about producing explanations of plans can be formulated, independently of the domains in which the plans are produced. This property, of providing a relatively formally defined and yet domain-independent input, makes plans very attractive from a natural language generation point of view.

This paper discusses a system that accepts a plan structure of the sort generated by AI planning programs and produces natural language text explaining how to execute the plan. Our objective in building this system has been to develop a clear model of a possible architecture for a language generation system that makes use of *simple, well-understood,* and *restricted* computational techniques. We feel that too much of the work in this area has been characterized by the use of arbitrary procedures, which often do not provide a clear basis for future work. We believe that by providing a simple yet nontrivial account of language generation, we can contribute at least by providing a "straw man" with known limitations, with respect to which other work can be compared.

Describing plans represents in many ways an obvious

Copyright 1989 by the Association for Computational Linguistics. Permission to copy without fee all or part of this material is granted provided that the copies are not made for direct commercial advantage and the *CL* reference and this copyright notice are included on the first page. To copy otherwise, or to republish, requires a fee and/or specific permission.

0362-613X/89/010233-249-$03.00

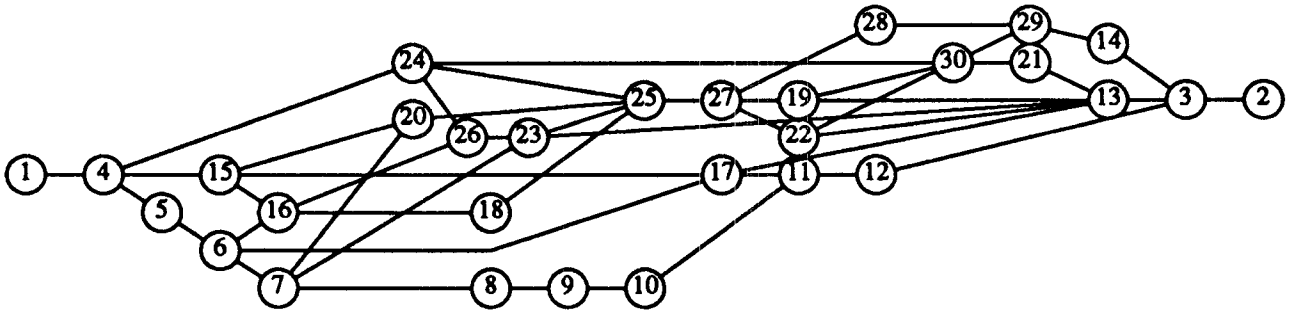


Figure 1 Action Graph of a Nonlinear Plan.

application of natural language generation, and our approach has been to tackle this problem in a fairly straightforward way, informed by the state of the art as we perceive it. The results from our system are promising, but our texts lack much of the smoothness of human-generated explanations. An analysis of the reasons behind some of the system's failures points to a number of deep problems concerning the connection between AI plans and natural language explanations.

In the next section we briefly introduce the inputs and structure of the language generation system. We then run through the parts of the system by showing a worked example. The core of this paper concerns the mapping from plans to *messages*, which can be thought of as abstract descriptions of natural language discourses. We describe our repertoire of messages, how plan structures are mapped onto messages, and how messages are simplified. Finally we look at further examples of the system's output and analyze some of its failures and successes.

2 System Overview

2.1 PLANS AND PLANNERS

For this project, we have adopted a traditional AI view of planning and plans. According to this view, the task of *planning* to achieve a goal is that of finding a set of (instantaneous) actions which, when performed, will transform the world as it is (the "initial state") to a new world (the "final state"), which is similar to the present world, but where in addition the goal is true.

We assume that the plans produced by our planner are *nonlinear* (almost standard with AI planners since NOAH; Sacerdoti 1975); that is, they only partially specify the order in which the actions are to be performed. Furthermore we assume that the time constraints involved in a plan can be displayed in an *action graph*, where an action is represented by a point and a line going rightward from one action to another indicates that the first action must take place before the second (this is true of most, but not all, AI plans—see for instance Tsang 1986). Figure 1 shows an action graph for a nonlinear plan for building a house.

We further assume that plans are in general *hierarchical*. By this we mean that the planner operates in a hierarchical manner (almost standard in AI planners since ABSTRIPS; Sacerdoti 1973), first producing a plan specified at a very abstract level, and then successively refining it until the required level of detail is obtained. At each stage a process of *criticism* may impose new orderings between actions whose relative ordering seemed to be unconstrained at the previous levels of abstraction. For us, the history of this hierarchical expansion must be present in the final plan, since we assume no explicit interaction with the planner itself while it is operating. (We shall return in Section 5 to the question of whether the hierarchical plan structure is in fact a sufficient description of the planner's processing.)

For concreteness, we have based our system on the output of a single AI planning program, even though there are a number of planning systems that could produce a similar style of output. The input to our natural language generator, then, is the translation into Prolog of the set of datastructures created by Tate's (1976) NONLIN planner.

2.2 SYSTEM STRUCTURE AND PARAMETRIZATION

We have set ourselves the goal of generating from a NONLIN plan a single natural language text that explains the actions to be performed and why things have to be done this way. To a large extent the explanatory power of such an account depends on what information is represented in the plan in the first place. Although a system that produces a single monolog from a plan is more restricted than, say, an interactive system that can be asked to explain parts of the plan selectively, a number of possible applications do suggest themselves (for instance, the automatic generation of instruction manuals), and the monolog task does provide us with an excellent way of studying the problems of automatically generating large texts.

We have attempted to factor out domain-dependence as much as possible in the generation system by having it rely heavily on knowledge expressed in a declarative fashion. Given a particular target natural language, a specific lexicon then needs to be provided for the domain in which the plans are to be generated (we have considered cookery, house building, car maintenance,

central heating installation, and the "blocks world"). This provides linguistic representations corresponding to the objects, states, and actions that will arise in generated plans. These lexical representations are supplemented by domain-dependent rewrite rules that can be used to reveal hidden additional structure in the planner's representation of the domain. Even with the target natural language fixed and a particular domain given, there are still in general many possible plans from which natural language could potentially be generated (indeed, many man-years of AI research were devoted to developing plans simply in the "blocks world").

Our natural language generation system can be thought of as consisting of four processing stages, centering on the construction and manipulation of an expression of our special *message language*, as follows:

Message Planning
Message Simplification
Compositional Structure Building
Linearization and Output

Message Planning is the interface between the generator and the outside world. At this stage, the generator must decide "what to say," i.e., which objects and relationships in the world are to be expressed in language and in roughly what order. The output of the message planner is an expression in the message language which, following McDonald, we will call the *message*. The idea is that message planning may be a relatively simple process and that the resulting message is then "cleaned up" and simplified by localized rewrite operations on the expression ("Message Simplification").

The message is a nonlinguistic object, and the task of *structure building* is to build a first description (a functional description much as in Functional Grammar; Kay 1979) of a linguistic object that will realize the intended message. We assume here that a "linguistically motivated" intermediate representation of the text is of value (this is argued for, for instance, by McDonald 1983). Our structure builder is purely *compositional*, and so the amount of information that it can take into account is limited. We treat structure-building as a recursive descent traversal of the message, using rules about how to build linguistic structures that correspond to local patterns in the message. During this, a simple *grammatical constraint satisfaction* system is used, to enforce grammaticality and propagate the consequences of syntactic decisions. The recursive descent terminates when it reaches elements of the message for which there are entries in the system's lexicon.

Once a structural description of a text has been produced, it is necessary to produce a linear sequence of words. Our structural descriptions contain only dominance information and no ordering information, and so a separate set of rules is used to produce a linearization. This is akin to the ID/LP distinction used in GPSG (Gazdar et al. 1985).

The resulting system is similar to McDonald's (1983) model, in that it is basically a direct production system that utilizes an intermediate syntactic representation. The system is also similar to McDonald's in its emphasis on local processing, although there is no attempt to produce a psychological model in our work. Our constraint satisfaction system is implemented efficiently by unification, however, so that the effects of local decisions can propagate globally without the need for explicit global variables. This is used, for instance, to enforce a simple model of pronominalization (based on that of Dale 1986).

3 A Worked Example

As an illustration of the various mechanisms contained within the system, we present in this section an example of the system in operation. The example is taken from a demonstration application, showing how the language generator might be attached to an expert system. The scenario is as follows: we have an expert system whose function is to diagnose car-starting faults. The expert system asks questions to which the user can either give an answer or type "how," meaning "how can I find out the answer?" In this latter case, the expert system invokes a planner to work out what the user has to do, and passes the resultant plan to the language generator, which produces text giving instructions to the user. The expert system then asks its original question again.

In our demonstration system, the expert system is in fact just a binary decision tree. At each internal node there is a yes-no question and a planner goal, to be used if the user responds with "how." At each leaf node there is a recommendation and a planner goal—here "how" is interpreted as "how do I carry out your recommendation?" To make the demonstration more varied, the system keeps track of what it has already told the user to do, so that, for example, accessing the carburetor jet will be described differently depending on whether the air filter (which is on top of the carburetor) has already been checked.

We pick up the example at a point where it has been ascertained that the battery is OK, but that there is no spark on the spark plugs. The next step is to test for a spark at the distributor. The system asks:

```
Is there a spark at the distributor?
```

and we respond with "how." The NONLIN plan goal associated with the above question is

```
{tested dist_spark}
```

that is, "make a plan to achieve the state in which we have tested the distributor spark." The planner assumes that we have done nothing already and are standing at the front of the car, looking at the engine.

3.1 The Plan

The plan produced by NONLIN for this example case is a totally ordered sequence of six actions as follows:

{act {detached dirt_cover engine}}	*detach the dirt cover from the engine*
{act {detached coil_lead dist_cap}}	*detach the coil lead from the distributor cap*
{act {located mech cab}}	*go to the cab*
{act {started engine}}	*start the engine*
{act {located mech frontofcar}}	*go to the front of the car*
{act {observed spark coil_lead}}	*observe whether there is a spark on the coil lead*

However, although this is the plan at its lowest level, the plan structure returned by NONLIN also includes the hierarchical expansion history of the plan. The plan started out as just the original goal itself, and was successively expanded to greater levels of detail until the primitive actions given above were obtained. The expansion hierarchy for this plan is shown in Figure 2.[2]

As well as this hierarchical structure (and the ordering information not shown in this diagram), NONLIN returns information about preconditions in the plan—where they are needed and where they are established. So, for example, the condition {goal {located mech cab}} is required by node 14 and made true by node 13 (and made false again by node 15).

3.2 THE TEXT

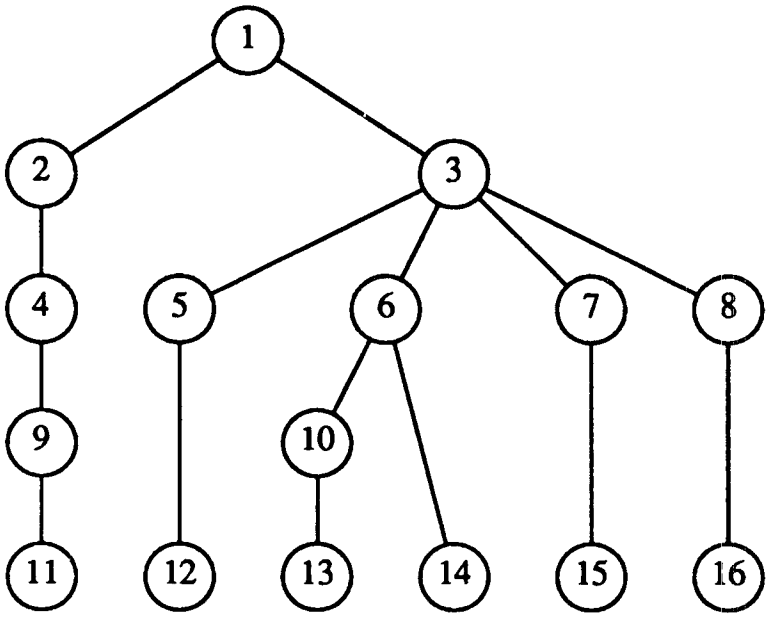
All this information is extracted from NONLIN's data structures, converted into Prolog clauses, and passed to the language generator. The generator looks for ways to break up and organize the information in the plan to produce comprehensible text. This process is described in more detail in Section 4, but to see what it does in this case, we shall concentrate on just one fragment of the above plan, namely nodes 3, 5, 6, 7, and 8. These nodes represent the expansion of the following NONLIN operator:

```
actschema tested_4
      pattern {act {tested dist_spark}}
      expansion
            goal {goal {detached coil_lead dist_cap}}
            goal {goal {started engine}}
            goal {goal {located mech frontofcar}}
            goal {goal {observed spark coil_lead}}
      orderings
            1 → 4
            2 → 4
            3 → 4
      conditions
            unsupervised {goal {accessible dist}} at self
            supervised {goal {detached coil_lead
              dist_cap}} at 4 from 1
            supervised {goal {located mech frontofcar}}
              at 4 from 3
            supervised {goal {started engine}} at 4 from
              2
end;
```

This operator expands the high level action "do something that causes the distributor spark to have been tested" (that is, "test the distributor spark") into four subgoals (not actions, since if they are already in effect, nothing further needs to be done), the first three of which must preceed the last. Thus the plan here is to ensure that the coil lead is detached from the distributor cap, the engine is started, and the mechanic is at the front of the car, and then to observe whether there is a spark on the coil lead.

The subplan gives rise to the following piece of text:

TESTING THE DISTRIBUTOR SPARK
Testing the distributor spark involves detaching the coil lead from the distributor cap, starting the engine, going



1: {goal {tested dist_spark}}
2: {goal {accessible dist}}
3: {act {tested dist_spark}}
4: {act {accessible dist}}
5: {goal {detached coil_lead dist_cap}}
6: {goal {started engine}}
7: {goal {located mech frontofcar}}
8: {goal {observed spark coil_lead}}
9: {goal {detached dirt_cover engine}}
10: {goal {located mech cab}}
11: {act {detached dirt_cover engine}}
12: {act {detached coil_lead dist_cap}}
13: {act {located mech cab}}
14: {act {started engine}}
15: {act {located mech frontofcar}}
16: {act {observed spark coil_lead}}

Figure 2 Expansion Hierarchy.

to the front of the car and then observing whether the coil lead sparks.

If you go to the front of the car now you will not be at the wheel afterwards. However in order to start the engine you must be at it. Therefore before going to the front of the car you must start the engine.

If you start the engine now you will not be at the front of the car afterwards. However in order to detach the coil lead from the distributor cap you must be at the front of the car. Therefore before starting it you must detach the coil lead from the distributor cap.

Detach the coil lead from the distributor cap. After this start the engine. After this go to the front of the car. After this observe whether the coil lead sparks.

You have now finished testing the distributor spark.

Notice that this text is not just a description of the actions as specified by the plan operator. Nor is it the fully detailed plan of everything to be done. It is a description of the plan operator in the context of the current plan, embellished with additional useful information from that context. It includes references to actions at several different levels of abstraction, as well as information about ordering constraints present in the plan but not present in the basic plan operator.

3.3 THE MESSAGE

The first step in the generation of this text is to convert the plan data into an expression in the generator's intermediate message language. The message language and the strategies for carrying out this conversion are discussed more fully in Section 4; here we concentrate on those aspects particularly relevant to this text.

The overall strategy applied to our subplan is to construct an *embedding*: an introduction-body-conclusion structure in which the introduction explains how the action is expanded, the body explains how to execute the expansion, and the conclusion makes the point that by executing the expansion, the higher level action is acheived. This strategy is appropriate because the body of the expansion is relatively simple. For more complex examples, where it is not practical to attempt to describe the expansion merely as an introductory sentence of a paragraph, an alternative strategy would be employed.

This strategy decision gives us the general shape of the text, and the components of the embedding are straightforwardly constructed, by reference to the local "shape" of the plan fragment. Here the actions are linearly ordered, which suggests presenting them in the order of execution. At the same time the message is embellished with the justifications for the action ordering. In the above plan operator, the first three actions were unordered, but lower level considerations (concerning where the mechanic is at a given time) impose an ordering on them in the actual plan returned. Message elements are added to explain these ordering requirements, and in this case these necessarily appeal to the lower level actions of moving about.

The resulting message expression is too large for easy display, so we shall concentrate on a small part of it, the part corresponding to the three sentences:

If you go to the front of the car now you will not be at the cab afterwards. However in order to start the engine you must be at it. Therefore before going to the front of the car you must start the engine.

The initial message expression for this text is:

```
implies (
    contra_seq (
        hypo_result (
            user,
            achieve (goal (located (mech, front ofcar))),
            not (goal (located (mech, cab)))),
        prereqs (
            user,
            then (wait ([ ]), achieve (goal (started
            (engine)))),
            goal (located (mech, cab)))),
    neccbefore ( user,
        then (wait ([ ]), achieve (goal (started
        (engine)))),
        achieve (goal (located (mech, frontofcar)))))
```

where expressions like goal(located(mech,frontofcar)) and goal(started(engine)) are straight NONLIN expressions, translated literally into Prolog. This expression can be read approximately as "the hypothetical result of going to the front of the car is that you will not be in the cab, and this contrasts with the prerequisite of being in the cab to start the engine. This combination implies you should start the engine before you go to the front of the car." And of course, that is more or less what the produced text says.

3.4 SIMPLIFYING THE MESSAGE

The above message contains a number of redundancies, which will lead to inelegant text if it is used for generation. First of all, it contains various occurrences of wait([ ]) (the action of waiting for nothing). These are inserted because in general at certain points of the subplan being explained, one is forced to wait for the conclusion of actions being performed in other subplans; this time, however, there are no such critical actions in other parts of the plan. Second, the NONLIN expressions have been inserted verbatim, without any consideration of whether they could be expressed more elegantly in the message language. Message simplification concerns rewriting the message expression generated by message planning into one that is "simpler" in some sense. This is achieved by applying rewrite rules to components of the whole message. Two kinds of rewrite rules are used: the first kind perform domain-independent structural simplifications to message expressions, and the second domain- or language-dependent alterations. For example, the following two rules together dispose of the redundant wait([ ]) terms of the above message ([ ] denotes the empty action here):

```
wait([ ]) → [ ].
then([ ],X) → X.
```

These are to be read as rewrite rules, with the expressions on the left of the "→" being rewritten to the expressions on the right. Variables are denoted by names beginning with capital letters. The operation of these rules is entirely domain-independent. One of the domain-dependent rules rewrites mech (the mechanic) as user, to indicate that the mechanic is the same as the person to whom the instructions are being given. Another set of rules rewrites states into a form where the affected person or object is explicit (this representation allows the system to collapse together multiple states involving the same person):

```
goal (located (X,Y)) → state (X, located (Y)).
```

More substantial examples include the following rules for talking about moving around:

```
achieve (state (user, located (Y))) → go_to (user,Y).
result (go_to (X,Y),state (user, located (Y))) →
do (go_to (X,Y)).
```

The first causes a phrase such as "get to be at the front of the car" to be rewritten as "go to the front of the car." The second removes redundancy in a sentence like "Go to the front of the car and you will be at the front of the car," rewriting it as simply "Go to the front of the car."

Once all the rewrite rules have been applied, our simplified example message looks like this:

```
implies (
    contra_seq (
        hypo_result (
            user,
            go_to (user, function (front, car)),
            state (user, not (located (cab)))),
        prereqs (
            user,
            start (engine),
            state (user, located (cab)))),
    neccbefore (user,
        start (engine),
        go_to (user, function (front, car))))
```

3.5 COMPOSITIONAL STRUCTURE BUILDING

The next stage is to build a linguistic structure from this message expression. The structure-building component uses an ordered set of rules for mapping from local message patterns to linguistic structures. It is similar to the system described in Chester (1976) in the local nature of its operation, but Chester builds sentences directly, rather than via structural representations. Mann et al. (1982) would call our system a "direct translation" system. A system built in this fashion has the advantage of a very simple control structure and has the potential of having its principles expressed in modular, independent rules.

Our linguistic structural descriptions are similar to the functional descriptions used in Functional Grammar (Kay 1979; Kay 1984). For example, the following is a slightly simplified version of the rule used to realize the *hypo_result* construct above:

```
hypo_result (Agent, Act, State) →
  [samesentence,
   conjn = [root = 'if'],
   first =
     [s,
      agent = [$Agent],
      pred = [active,
      morph = pres,
      $Act,
      adv =+ [ap, adv_word=[root=now]]]],
   rest =
     [s,
      pred =
         [vp,
          aux = [root = will],
          pred =
             [vp,
              $State,
              morph = inf,
              adv =+ [ap,
                      adv_word=[root=afterwards]]]]]].
```

In this rule, the left hand side of the → is a Prolog pattern to be matched with part of the message (symbols beginning with uppercase letters represent variables, which are to match subexpressions of the message). The righthand side is a functional description, describing the English phrase that is to render that part. In these functional descriptions, expressions preceded by dollar signs represent the places where further information will be contributed by the expansion of subparts of the message. Thus the "agent" value is obtained by recursively matching the value of the variable Agent (that is, the first argument in the hypo_result term) against the structure rules.

This rule is responsible for sentences like:

If you start the engine now you will not be at the front of the car afterwards.

The rule provides the basic template for the sentence: it is a combination of two sentences using the conjunction "if." The first sentence is present tense active, with agent specified by the Agent argument and predicate by the Act argument, and has an adverbial modifer "now." The second sentence is a future tense expression of the State given as argument, with adverbial modifier "afterwards." The presence of samesentencer ensures that the whole is a single sentence and that the two subclauses have the same focus.

The recursive structure building process "bottoms out" when a message element is reached for which a linguistic realization appears in the domain-dependent lexicon. Domain states and properties (Section 4.1) are provided with lexical entries that describe how to realize them as VPs. Such entries could be written as structure building rules in the same format as the above hypo_result rule, but in practice it is convenient to use a more compact notation:

```
lx (accessible, be, [attr: @ accessible]).
lx (answer, answer, [obj: @ 'the question']).
lx (start (Z), start, [obj: Z]).
```

These entries indicate how each of accessible (a property), answer, and start(Z) (actions) can be realized in English, by providing a verb and a specification for the complements to follow it. In the first two, the complement phrases are specified directly as constant strings (indicated by the '@' sign). In the last one, the filler of the obj role (the direct object) will be whatever phrase is used to realize the object Z being started. Additional rules provide possible fixed phrases to realize such domain objects:

```
referent (engine, @engine).
```

When a domain object like engine comes to be realized, either the fixed phrase provided (prefixed by 'the') will be used, or a pronoun with the appropriate gender and number will be chosen. It is clearly a limitation of our system that no other possibilities are currently allowed, but in some sense this reflects the fact that plans come with a great deal of information about the actions to be performed but very little information about the objects involved in them.

Structure building rules are *ordered,* so rules for more specific patterns can be placed before rules for less specific ones, without the latter having to explicitly provide negative conditions. In addition, rule application is *deterministic,* in that, once the left hand side of a rule has matched a piece of message and the right hand side structure (minus the parts that require recursive rule matching) has been successfully built, no other rule will ever be tried for that portion of the message.

As well as the usual specifications of features and values (for example, conjn and first used above), functional descriptions can also contain specifications of properties, such as s and samesentence, that the relevant construction must have. Some of these properties (such as s) are intrinsic—essentially just features without values. Others are "macros" for bundles of simpler feature-value and property specifications. For example, samesentence is defined as being shorthand for a bundle of feature-value pairs that limit the possibilities for focus movement in and around the structure described. A collection of such macros enables us to implement what is essentially Dale's (1986) model of how discourse structure constrains pronominalization, which was inspired by the work of Grosz (1977) and Sidner (1979). The use of a macro like samesentence (keyed off particular structures in the message) sets up an environment that will allow certain pronominalizations but exclude others. The choice of whether to pronominalize or not is then made on a local basis. It is interesting to compare this scheme to that of McKeown (1982), which also makes focus decisions on a local basis. McKeown's approach, almost the opposite to ours, is to take certain focus priorities as primary and then to attempt to select material in accordance with these. Our approach, which involves considering focus only after the material has already been organized, regards pronominalization more as a last-minute lexical optimization than as something that is planned in advance. We have considered incorporating some means of focus annotation in the message, but it is not always clear at this level what the focus should be. We have thus preferred to allow message planning simply to place general constraints on focus movement.

As the message is traversed by the structure building rules, more and more information accumulates about the output functional description and its components. As is usual in unification grammar, in the written form structural descriptions are *sideways open,* that is, an object satisfying the description is required to have the features listed, but may have any other features in addition. Thus our structure building rules only provide the framework of the final functional description. The rest is filled in by a simple *grammatical constraint satisfaction system.* This enforces grammaticality and handles the propagation of feature values that are shared between different phrases (for instance, number agreement). The constraint satisfaction system is based on the use of a declarative "grammar specification" of the types of legal descriptions and the constraints they must satisfy. This specification is compiled into a representation that essentially treats every property and feature as a macro for a bundle of conclusions that follow from its being involved in a description.

3.6 CONSTITUENT ORDERING

The final task once the linguistic structure has been built is to determine the order in which the constituents are to be produced, and to locate the actual words to be used. Substructure ordering is determined by ordering rules. The ordering rules are applied to a structural description in much the same way that structure-building rules are applied to the message; that is, recursively and compositionally. The left hand side of an ordering rule is a pattern that is matched against the structural description. The right hand side of the first rule whose pattern matches is then taken as a template determining which parts of the description are to be realized as phrases and in what order. For example, here is a rule for VP ordering:

```
[vp, mainverb = V, adv = A, compls = C] → [V,C,A].
```

This rule ensures that a verb is realized before its complements, which are realized before any adverbial modifiers, producing VPs like:

go to the front of the car now

Each application of an ordering rule returns an ordered list of functional descriptions. These are then recursively subjected to ordering rules, to determine their relevant subphrases and the order these should be realized in. The recursion "bottoms out" when a functional description of type word is reached. The end result is a list of word descriptions each containing features detailing aspects of the morphology. These are passed to the morphology component (currently ex-

pressed as raw Prolog code), which will then output the appropriately inflected word.

4 Plans and Messages

4.1 The Message Language

In some ways, a natural language generation system is like an optimizing compiler. Producing some sort of natural language from a symbolic input is not a task of great difficulty, but producing text that is smooth and readable is a challenge (and in general quite beyond the state of the art). With both tasks one has the option of planning the text and simplifying its form either in a single pass or in multiple passes. In language generation, McDonald's MUMBLE (McDonald 1983) produces and simplifies linguistic structures within a single pass of the input. Although the modeling of human language production may require a theory of this kind in the end, the result is a system where it can be hard to separate out the different principles of structure building and simplification, because these have all been conflated for reasons of efficiency. We have thus opted for a multi-pass system. Multi-pass optimizing compilers need to have specialized internal languages (virtual machine codes) more abstract than the output machine codes and in terms of which optimizations can be stated. The analog in a natural language generation system would be a *message language* that could express at a level more abstract than linguistic structure the goals and intended content of a piece of language to be generated. We can see elements of such a language in the "realization specifications" of McDonald and Conklin (1982) and in the "protosentences" of Mann and Moore (1981). A crucial part of our own system is the use of a message language specialized for the explanation of plans.

Our message language is a language specifically devised for expressing the objects that arise in plans and the kinds of things one might wish to say about them. The main types of statements ("utterances") that can be made at present as part of our generated text are shown in Figure 3. These "utterances" mention actions and states, which could be domain actions and states (as appearing in the plan) or complex actions and states, formed according to the rules in Figure 4. The message language provides for the description of actions being carried out involving different agents and objects (both represented as "objects"—Figure 5), although NONLIN provides no indication about who is responsible for any given part of a plan. Thus the agent of an action defaults to user for an action that is properly part of the current subplan and someone for an action that has been included in the description but is properly part of another subplan. In this way, each part of the plan is explained from the point of view of the person executing it, with no assumption that the same person will be executing other parts of the plan. A message consists of a number of "utterances" linked together by various

```
UTTERANCE ::=
   neccbefore(OBJECT,ACTION,ACTION)
   --- one action must take place before another
   do(ACTION)
   --- instruction to perform an action
   result(ACTION,STATE)
   --- as 'do', but also mentioning an effect of the action
   hypo_result(OBJECT,ACTION,STATE)
   --- if the agent carried out the action, the state would hold
   expansion(ACTION,ACTION)
   --- describing the expansion of an action into subactions
   prereqs(OBJECT,ACTION,STATE)
   --- describing the prerequisites of an action, with the
   --- assumption that a given agent will perform it
   needed(OBJECT,ACTION,STATE)
   --- describing the reason why a STATE is needed, so that
   --- OBJECT can perform ACTION
   causes(STATE,STATE)
   --- once the first state holds, so does the second
   now(STATE)
   --- indicating that some state now holds
```

Figure 3 Types of Basic Utterance.

organizational devices. These indicate various kinds of sequencing and embedding (Figure 6). Most are simply ways to string together two "utterances," with an appropriate conjunction being suggested, according to what kind of link there is between the two. The embed construction is used to indicate a discourse segment which has an introductory section, a body and a concluding section. Hence it has three parts. The idea is that the explicit marking of such structures in the message language will enable linguistic decisions (for instance concerning pronominalization) to be made more intelligently. In general, the domain-dependent lexicon need only supply a single linguistic representation for the simplest form of a domain action or property. The linguistic forms of the more complex forms allowed by the message language are then dealt with automatically by the system (Figure 7).

4.2 From Plan to Message

A plan with 30 or so actions contains a great deal of material, spelling out the necessary partial ordering between the actions and their preconditions and effects. A crucial task in message planning is cutting this material down into small enough pieces that can be rendered as independent pieces of text. In a domain-independent system for plan explanation, the only structure that such a "chunking" can make use of is the abstraction hierarchy and the local "shape" of the action graph. Even this is unfortunately limited by the fact that the abstraction hierarchy may represent a view of the domain that is convenient to the plan generator, but not the plan executer.

The abstraction hierarchy tells us how certain actions

```
ACTION ::=
    then(ACTION,ACTION)
    --- two actions in sequence
    achieve(STATE)
    --- the action to achieve a state
    wait(STATE)
    --- waiting until a state holds
    complete(ACTION)
    --- finishing doing a prolonged action
    repeat(ACTION,STATE)
    --- doing the action until the state holds
    delegate(OBJECT,ACTION)
    --- have someone else do an action
    parallel(ACTION,ACTION)
    --- doing two actions in parallel
    DOMAIN_ACTION

STATE ::=
    and(STATE,STATE)
    --- both states hold
    state(OBJECT,PROPERTY)
    --- an agent has a property
    not(state(OBJECT,PROPERTY))
    --- an agent does not have a property

PROPERTY ::=
    andp(PROPERTY,PROPERTY)
    --- conjunction of properties
    enabled(ACTION)
    --- able to perform an action
    done(ACTION)
    --- having done an action
    doing(ACTION)
    --- doing an action
    DOMAIN_PROPERTY
```

Figure 4 Actions, States, and Properties.

at a particular level of the plan arise from the expansion of a single action at a more abstract level. Such a group of actions is an obvious candidate for explaining as a single chunk. Thus our basic strategy is to first of all talk about the plan at the most abstract level, then discuss the expansion of the actions at that level, then discuss the expansion of the actions involved in that, and so on. In general, then, at any time we are concerned with

1) selecting out the portion of the plan that corresponds to the expansion of a single abstract action and

```
OBJECT ::=
    user
    someone
    function(FWORD,OBJECT)
    DOMAIN_OBJECT
```

Figure 5 Objects.

```
MESSAGE :==
    title(ACTION,MESSAGE)
    --- labels a piece of text with a title (based on an action)
    embed(MESSAGE,MESSAGE,MESSAGE)
    --- introduction-body-conclusion type structure
    neutral_seq(MESSAGE,MESSAGE)
    --- two bits produced in sequence, but with no implied relationship
    time_then(MESSAGE,MESSAGE)
    --- two bits produced in sequence, this indicating time order
    linked(MESSAGE,MESSAGE)
    --- two bits produced in sequence, with some unspecified relatinship
    time_parallel(MESSAGE,MESSAGE)
    --- two bits produced in sequence, indicating parallelism in time
    contra_seq(MESSAGE,MESSAGE)
    --- two bits produced in sequence, where the second contradicts an
    --- expectation created by the first
    implies(MESSAGE,MESSAGE)
    --- one dplan statement is true and hence so is another
    UTTERANCE
```

Figure 6 Linking Devices for Utterances.

2) describing this, given that whole subsets of the actions in it are to be treated as single actions.

The first of these is not trivial because, as the result of successive criticisms, the set of actions in the expansion of a more abstract action may no longer be a simple connected piece of the plan. As an example of this, Figure 8 is the action graph for a house-building plan, with the actions that are in the expansion of "installing the services" blocked out.

To describe this set of actions and their timing in the plan, it is necessary to describe other actions whose timing is closely coupled to them. The actions to be included in the explanation of the expansion are obtained by a "closure" operation—a process of tracing through all possible paths going forward in time between actions in the expansion. Any other actions encountered on these paths are deemed necessary to be included in the description. We call these actions "intruders." Thus the actions described form the minimal convex graph that includes the desired actions (Figure 9).

Once the lowest-level plan actions corresponding to a single abstract action have been isolated, the "shape" of this part of the plan at the current level of abstraction needs to be determined. The current point in the abstraction hierarchy specifies the set of actions that can be mentioned in this part of the text. If one of these is an abstract action, in general there will be a whole class of lowest-level actions that need to be described simply as parts of this action, whose internal structure will be described later. The lowest-level actions are thus grouped into subsets, and what is to be explained are relationships between these subsets, rather than relationships between primitive actions. Technically, the "chunking" imposed by the current layer of the abstraction hierarchy defines an equivalence relation, and

do(examine(filter_bolts)).	**Examine the filter bolts.**
do(complete(examine(filter_bolts))).	**Finish examining the filter bolts.**
do(delegate(someone,examine(filter_bolts))).	**Have someone examine the filter bolts.**
now(state(user,enabled(examine(filter_bolts)))).	**You can now examine the filter bolts.**
now(state(someone,done(examine(filter_bolts)))).	**The filter bolts have now been examined.**
now(state(user,doing(examine(filter_bolts)))).	**You are now examining the filter bolts.**
now(state(plug_leads,positioned)).	**The plug leads are now in position.**
do(achieve(state(plug_leads,positioned))).	**Get the plug leads to be in position.**
do(wait(state(plug_leads,positioned))).	**Wait until the plug leads are in position.**
causes(state(user,done(achieve(state(plug_leads,positioned)))),	**Once the plug leads are in position you can**
state(user,enabled(examine(filter_bolts))))).	**examine the filter bolts.**

Figure 7 Expressions Generated Using Two Lexical Entries.

we are interested in the *quotient plan* with respect to this relation. We can define the usual plan relationships between the relevant subsets of actions in a natural way. For instance, we say that one subset comes before another if and only if each element of the first comes before each element of the second. Because such a demanding criterion will apply quite rarely, in general there will be a great deal of parallelism in subplans whose actions are not at the most detailed level.

Once a piece of the whole plan has been extracted and its "shape" (relative to some given equivalence relation) established, rhetorical strategies are applied to decide how particular parts are to be presented. The message created depends directly on the structure of the justified plan. Thus, for instance, the expansion of a complex action gives rise to a section of text represented by a message of the form:

```
title ( Action,
        embed ( Intro,
                Body,
                now (state (user, done (complete
                (Action)))))))
```

where Action is the action described, Intro is an introductory message, which describes the prerequisites of the main action and the set of actions in its expansion (unless there are too many of them) and Body describes the action graph expanding the main action. The main strategies for describing action graphs are the *lump strategy, forwards description*, and *backwards description* (Figure 10). The lump strategy applies if a piece of the action graph is a self-contained "lump" between two actions A and B, with no links between any actions inside the "lump" and any actions outside. If the subgraph between the actions is sufficiently complex (has more structure than two simple actions in parallel), the strategy suggests that its explanation should be postponed to a separate "section." Meanwhile the whole subgraph is incorporated into the current explanation as if it were a simple action (this is, of course, the same strategy that is applied for an action that is above the primitive level in the abstraction hierarchy). Forward description is deemed appropriate when the action graph is a right-branching structure; in this case the actions are generally dealt with in time order, giving a message of one of the forms:

```
time_then (result (Act, State), ...)

neutral_seq (
    result (Act, State),
    embed (causes (State,
                   state (user, enabled (Acts))),
           ..., [])
```

where Act is the first action, State a state that it makes true, and ". . ." is the message derived from the subsequent actions. When the action graph is a left branching structure, however, the strategy of backwards description is suggested. This gives rise to messages of the form:

```
embed (prereqs(user,Act,Pres),
       ...,
       now (state(user,
                  enabled (
```

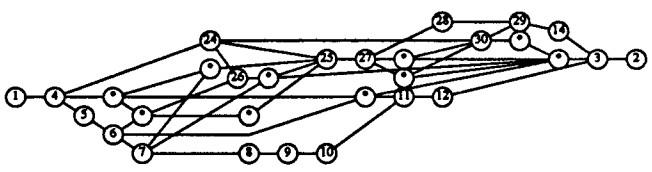


Figure 8 Distribution of Expression of an Abstract Action.

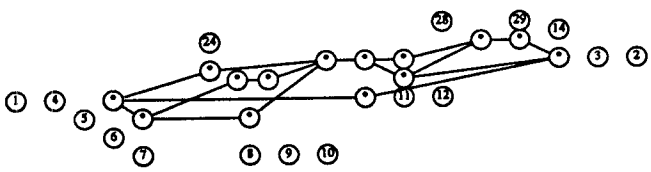


Figure 9 Closure of Expansion.

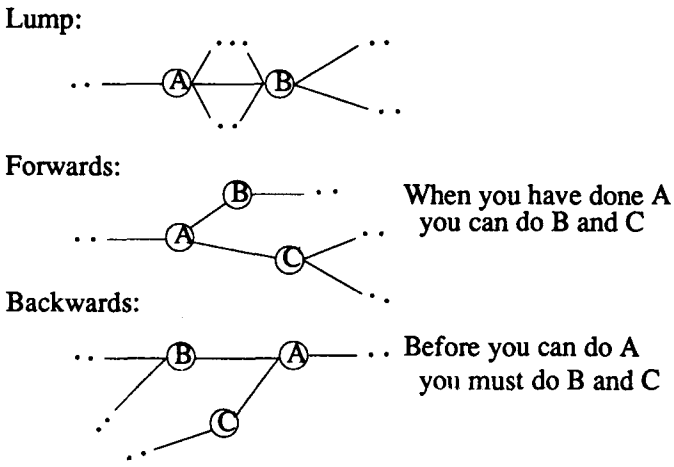


Figure 10 Rhetorical Strategies.

```
parallel (Act,
          achieve (State)))))
```

where Act is the first action, with preconditions Pres and effects State, and ". . ." is the message derived from the subsequent actions.

All of these kinds of messages require the insertion of preconditions and effects of actions. It is necessary for the system to compute those preconditions and effects that are actually relevent for the current plan, rather than simply the total sets of preconditions and effects. This amounts to determining the justifications for the action ordering chosen. The justification for action A coming before action B can be of one of two types. Either A is needed to create a state where a precondition of B is true, or A comes before B because otherwise B would create a state where a precondition of A was not true. The two different possibilities give rise to different modes of presentation, but if the justification is redundant or not available from the plan, it is simply missed out.

4.3 IMPROVING THE MESSAGE

As the last section suggests, the initial version of the message is put together in a very direct way from the structure of the plan. As a result, it is often unnecessarily cumbersome. Message simplification concerns rewriting the message expression generated by message planning into one that is "simpler" in some sense. Since the amount of material we wish to deal with could be large, we have avoided considering expensive global simplification techniques in favor of emphasizing local simplification techniques analogous to "peephole" optimizing techniques in compiling. Of course, a crucial difference between language generation and compilation is that in the former there is no clear notion of what "optimality" is. In the absence of a formal and detailed psychological theory of discourse comprehension, researchers in natural language generation are reduced more or less to using their intuitions about whether one way of phrasing something is "easier to understand" than another. We have regretfully had to follow the same course in designing and evaluating our own system.

The domain-independent simplification rules used by our message simplification system are treated equally, but conceptually they seem to be of four main types. Members of the first type tidy up special cases that could as easily be detected when the expression is constructed. Here is an example of such a rule ([] denotes the empty utterance):

```
(1) neutral_seq (X,[]) → X.
```

Thus any utterance expression of type neutral_seq will be rewritten by this rule if its second component is empty. Such an expression is rewritten simply to its first component. Incorporating such rules into the simplification stage means that the message-planning component can be simpler and more perspicuous.

The second kind of rule expresses knowledge about planning and plan execution. Here are two such rules:

```
(2) achieve (state (user, done (Act))) → Act.
(3) parallel (X, wait (Y)) → then (X, wait (Y)).
```

Rule (2) expresses the fact that the only way to create a state where you have done an action is to do the action. Rule (3) expresses the fact that waiting is an action that is always postponed until there is nothing else to do. Both of these principles are useful in finding the best way to express a given action.

A third kind of rule really reflects the linguistic coverage of the system in an indirect manner. If there is a special way available for saying a particular kind of thing, then that should be preferred to using a more general technique. Here is such a rule:

```
(4) prereqs (user, X, state (user, done (Y))) →
    neccbefore (user, Y, X).
```

This rule is about a special case of the prereqs structure arising in the message. When one is calling attention to the prerequisite(s) of an action X, a special case arises when the only prerequisite is the achievement of another action Y. In this case, the prerequisites statement amounts to saying simply that Y must happen before X. In general, one would expect that expressing the statement in this second way would result in a simpler piece of text than using a general-purpose strategy for expressing prereqs statements. It is arguable that such rules should really exist as special-case structure building rules. Such an approach would, however, preclude the use of simplification rules that made further use of the output of such rules.

Finally, there are rules that are motivated by notions of simplicity of structure. For instance, the rule:

```
(5) time_parallel (do (X), do (Y)) → do (parallel (X,
Y)).
```

results in an expression with one fewer "connectives." Such rules should really be backed up by a (perhaps psychological) theory of the complexity of messages.

Here is an example of how a message language expression can be simplified using these rules.

```
neutral_seq (
   prereqs (user,
            achieve (state (user, done (a1))),
            state (user,
                   done (parallel (a2, wait (s)))),
            [])
```

is simplified by rule (1) to:

```
prereqs (user,
         achieve (state (user, done (a1))),
         state (user,
                done (parallel (a2, wait(s)))))
```

which is simplified by rule (2) to:

```
prereqs (user,
         a1,
         state (user, done (parallel (a2, wait(s)))))
```

which is simplified by rule (3) to:

```
prereqs (user,
         a1,
         state (user, done (then (a2, wait(s)))))
```

which is simplified by rule (4) to:

```
neccbefore (user, then (a2, wait(s)), a1)
```

Here the simplification would result in the difference between a text like:

In order to get you to have washed the baby you must have undressed the baby and waited until the bath is full.

and one like the following:

You must undress the baby and then wait until the bath is full before you can wash the baby

The rewrite rules we have discussed so far in this section are independent of the domain in which the plan is made. Our system also allows for domain-dependent rules to be provided for a given planning domain. This provides a way of automatically rewriting every occurrence of a given expression coming from the planner into another given expression. One purpose of this kind of rule is to provide a translation for states, which may be primitive objects to the planner but are required to be somewhat more complex by the generator. For example, in the car domain, there is a rule that rewrites the planner primitive positioned (X) to be the complex term state (X, positioned). Domain-dependent rewrite rules can also be used to show correspondances between action and state names that seem independent but are in fact strongly connected. For instance, in the house-building plan, there is an action lay_basement_floor and a domain state basement_floor_laid (not a legal message state). Not surprisingly, the second is an effect of the first and can only come about by the first having been done. Given that we can deal with complex states and actions, we would do well to replace the second by a formula involving the first, in fact state (user, done (lay_basement_floor)). In this way we can simplify certain expressions in the message. For instance, the expression:

```
do (achieve (basement_floor_laid))
```

is equivalent to:

```
do (achieve (state (user, done (lay_basement_floor))))
```

which simplifies to:

```
do (lay_basement_floor)
```

by simplification rule (2) above. Given this domain-dependent rule and the simplifications thus enabled, the expression do (achieve (basement_floor_laid)) would be realized as something like "lay the basement floor," rather than "get the basement floor to be laid," which would arise from a more straightforward encoding of the state basement_floor_laid in terms of verbs and cases.

Domain-dependent rewrite rules allow us, in principle, infinitely to enrich the semantics of actions and states represented in the plan. They thus provide one way of compensating for the shallowness of the planner's representation. The basic framework on which the plan actions and states hang is, however, fixed by the planner and cannot be changed by the generator. Thus not all deficiencies of the planner can be rectified by this method. The extensive use of domain-dependent rewrite rules is in any case unattractive, as it takes away from the domain-independence of the system. We will return to this topic later.

4.4 KNOWLEDGE SOURCES IN MESSAGE CONSTRUCTION

Before we leave our discussion of how messages are constructed, it is useful to summarize the different knowledge sources that have an effect on the text generated from a plan. The gross organization of the message is determined by rhetorical strategies that look for patterns in the plan structure. Such strategies are specific only to the kind of plan that we are taking as input (i.e., hierarchical, nonlinear plans). Message simplification is usually responsible for the finer-grain structure of the message, as its rewrite rules operate strictly locally in the message. The domain-independent rewrite rules exploit the redundancy in the message language and express heuristics about how a given proposition might be expressed most simply. Such rules embody simple knowledge about planning and the facilities of the message language. Finally, domain-dependent rewrite rules enable some of the hidden structure in the planner's representation to be revealed.

Once a final message has been decided on, its realization as text makes use of structure building rules that depend on the natural language being used. At this point most of the significant decisions have already been made. The structure-building rules are able to make a

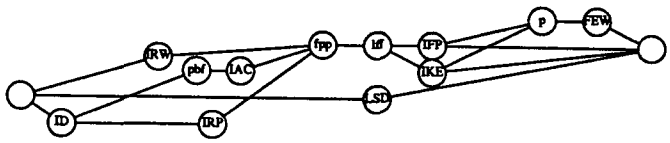


Figure 11 Installing the Services.

limited choice among possible syntactic structures and are able to introduce pronominalization where it seems appropriate, but their scope is heavily constrained by the message. During structure building, a domain-dependent lexicon makes available a verb entry for each domain state and action, as well as a fixed NP that can be used to denote each domain object. Although it is useful to assess the effectiveness of the system by considering the text output, many of the more interesting problems with the system are really already visible at the message stage.

5. Discussion

5.1 FURTHER EXAMPLES

The system has been tested using a number of different domains with rather different characteristics, and the results have been correspondingly varied. One domain that seems to work fairly well is that of cookery recipes such as the following:

MAKING PAPRIKA POTATOES AND SMOKED SAUSAGES
Melt the fat, fry the onion in it, add the flour to it and add the paprika to it. After this, stir the sauce until it boils. Meanwhile peel the potatoes and cut them into pieces. After this, add them to the sauce, cover the pan and make the sauce boil, stirring the sauce occasionally. Meanwhile cook the sausages. After this, add them to the sauce.

This text was actually produced from a "mockup" of plausible planner output, rather than a real plan, and did not include enough information (about preconditions, effects, etc.) to warrant the system adding justifications about ordering. This does not seem to matter too much, probably because cookery recipes are traditionally presented as instructions to be followed more or less blindly.

For an example where our techniques produce a less pleasing result, consider the "installing the services" extract from the house-building plan (discussed above) shown in Figure 11. In this action graph (which shows no preconditions or effects), we have indicated the actions with abbreviated names. Those actions in lowercase are not actually part of installing the services (but are "intruder" actions that are nevertheless crucial to this part of the plan); they will be described elsewhere in the text. Here is the English produced for this plan fragment:

INSTALLING THE SERVICES
Installing the services involves finishing the electrical work and laying the storm drains.
You must paint the house before finishing the electrical work.
In order to paint the house you must have installed the finished plumbing and installed the kitchen equipment.
You must lay the finished flooring before installing the finished plumbing and installing the kitchen equipment.
You must fasten the plaster and plaster board before laying the finished flooring. In order to fasten the plaster and plaster board you must have installed the air conditioning and installed the rough plumbing and installed the rough wiring.
Install the drains and then install the air conditioning, installing the rough plumbing.
Meanwhile install the rough wiring,
You can now fasten the plaster and plaster board.
You can now lay the finished flooring.
You can now install the finished plumbing and install the kitchen equipment.
You can now paint the house.
You can now finish the electrical work.
Meanwhile lay the storm drains.
You have now finished installing the services.

This account is basically comprehensible, but is repetitive and quite hard to follow. One reason for the repetition is that the subject matter is really very boring and uninformative, and it would be quite a challenging task for a human being to produce interesting and readable text from the same information. We discuss below some other reasons why this text is less than optimal.

5.2 DEFICIENCIES IN PLANS

Although generating explanations from the output of an AI planner appears to be a promising application of natural language generation research, there are a number of special problems that we have encountered with this task. Indeed, we can explain some of the deficiences in the text we have been able to generate purely in terms of deficiencies of the planner and/or its plans. Some problems stem from the use of plan operators not designed with text generation in mind, and can be solved within the scope of the planning system. More serious are problems that arise because of deficiencies in the planner methodology itself. In the development of our system we have encountered a number of these, ranging from trivial to quite fundamental. Some of these are properties of NONLIN in particular; others apply more generally to most AI planning systems. It is not appropriate to discuss the full details of these problems here, but we shall mention some of the main points.

GRANULARITY

One might ask why, unlike in the cookery recipe, there is no pronominalization in the text for installing the services. The coherence of the text would be improved

considerably by the judicious use of pronouns. Unfortunately, whereas in the cookery domain (which we encoded by hand) a particular action of 'frying' is treated as an instance of a general action that can potentially be applied to different objects; in the house-building domain the objects acted on by an action are fundamentally built into that action. The difference can be seen from example lexical entries from the two domains:

```
lx (fry (Food, In), fry, [obj: Food, in: In]).
lx (install_rough_wiring, install,
  [obj : @ 'the rough wiring']).
```

To use pronominalization, one needs to be able to determine that the same domain object is being mentioned several times, but only the first type of representation here actually supports the representation of domain objects. What has happened here is that, from the point of view of making a house-building plan, the planner cannot make use of properties of a general action like 'install,' and so its representation of actions is at a coarser level of granularity than that required to produce good text.

In common with most plans in traditional AI work, NONLIN plans only encode very weak information about causality and temporal relationships. For instance, when there is an action that achieves an effect, there is no way to tell from the plan whether we are dealing with an instantaneous action, an extended action that terminates as soon as the effect is achieved, or an extended action where the effect is achieved sometime during the execution. A natural language like English provides ways of distinguishing between these cases:

Turn on the switch and the light will be on.
Pour in the water until the bucket is full.
Prepare a chicken curry so that the chicken scraps are used up.

Because there is no way to distinguish between these in the NONLIN representation of effects, our generator is forced to try to find a neutral way to express all of them. As a result, there is a homogeneity in the text that is not necessarily reflected in the actual plan execution. Again the problem can be thought of as a mismatch between the granularity of the representation used for planning and that needed to exploit the facilities of the natural language.

The effect of the granularity problem can be lessened by allowing the plan generator to provide deeper information about the internal structure of actions and states through domain-dependent rewrite rules. Our message language allows us to talk about repeated actions, for instance, and so we can specify that certain domain actions are really shorthand for more complex expressions:

```
fill_bucket → repeat (pour (water, bucket),
                      state (bucket, full)).
```

Messages containing these complex actions can then be simplified by domain-independent rules like:

```
result (repeat (Act, State), State) →
   do (repeat (Act, State)).
```

Similarly we can use domain-dependent rewrite rules to introduce tokens standing for domain objects and hence give us a basis for pronominalization. The more one relies on domain-dependent rewrite rules for good text, however, the less one can claim to have a domain-independent basis for generating text from plans.

CONCEPTUAL FRAMEWORK

Domain-dependent rewrite rules can be regarded as a way of embellishing the planner's output to match up better with the requirements for generation. The basic framework of the plan is, however, something that cannot be changed unless the generator itself is to start doing some of the planning. Assuming that there is some point in distinguishing the planner from the generator, the generator is therefore sometimes faced with a mismatch between self-evident concepts in the planner's conceptual framework and those concepts that can be expressed simply in natural language. Consider, for instance, the notion of (primitive) actions that are unordered in the plan. If two actions have no ordering relation between them, then this indicates that the actions can be performed in any order relative to one another. To express correctly the plan's semantics, one should therefore make use of expressions like:

Install the drains and install the rough wiring, in any order.

In practice, however, we have chosen to map such a piece of plan into a message like:

```
do (parallel (install_drains, install_rough_wiring)).
```

which then gives rise to a text such as:

Install the drains. Meanwhile install the rough wiring,

Treating unordered actions as parallel actions may indeed both produce good text and even capture the reality of plan execution, as in:

Make the sauce boil, stirring the sauce occasionally.

but this will only be so if at least one of the actions takes place over a period of time and the actions can be and are recommended to be executed concurrently. There is, of course, no way to determine from the planner's representation whether this is so. Indeed, since the planner regards all primitive actions as essentially instantaneous, *in all cases* it is in some sense incorrect to express the planner's recommendations in this way. If the correct execution of the plan were critical, for instance, then it could be very dangerous to hide the limited way in which the planner views the world as we have done. It might thus be suggested that a generator working from plans could and should always strive to convey the plan semantics accurately, even if this

involves long-winded and unnatural prose. But in the end one is faced with an incompatibility between the planner's conceptual framework and the limits of what our language can express. For instance, it unclear how the action of "installing the rough wiring" can be expressed in English in such a way that the action can *only* be interpreted as an instantaneous action, which is the way the planner sees it.

EXPLANATORY POWER

The texts that we have generated from plans are intended to do more than simply tell the reader how to execute a series of actions. We always hoped that the justification structure built by the planner would also help us to explain why the given actions, with the ordering described, are the right ones to achieve the plan's goal. In practice, however, our texts have failed to be explanatory for a number of reasons. One problem is that, unlike instructions generated by human beings, our texts only tell you *what* to do, and not *what not* to do. It is often just as important for a person to be warned about the unpleasant consequences of doing the wrong thing as it is to be told what the right thing is. Unfortunately, the notion of "plan" we have adopted only makes reference to the successful actions, even though the plan generator may have spent a lot of time exploring other possibilities that did not work out. It might therefore be appropriate, in future work, to consider natural language generation based on the *trace* of a planning system, rather than on the final result.

Similarly, in many of the texts produced by our system the reader is told *what* to do but is given no illumination as to *why* things have to be done in this way. Unfortunately, although in principle every plan is justified by earlier actions achieving the preconditions of later actions, many plans do not contain this information in a useful form—in the housebuilding plan, for instance, the only preconditions that are required for an action in this plan to be performed are the successful completion of previous actions. That is, the person who has encoded the operators in terms of which the plan is constructed has "compiled in" certain ordering constraints without using the language of preconditions and effects effectively to explain them. One is reminded here of the problems that Swartout (1983) encountered in producing explanations from expert systems. The problem was that just because a set of rules was sufficient to produce expert behavior did not mean that those rules contained anything illuminating to put into explanations. Similarly in the planning area, there is no reason why a set of operators that are effective for producing useful plans need contain anything very interesting that can be put into a natural language account. Unfortunately, one cannot necessarily expect machine-generated plans to come at the right level of detail to be really useful to a human being. For instance, a house-building plan that enabled one to see why the rough plumbing must be installed after the drains (presumably because otherwise it is hard to make the pipes line up) would be very large, and it would be well beyond the state of the art for such a plan to be produced automatically. Moreover, such a plan would undoubtably contain a lot of information that was blindingly obvious to a human reader and hence of no interest whatsoever.

ARBITRARY PLANNER RESTRICTIONS

As is typical with application programs, most planners have particular features that represent non-standard or novel approaches to certain situations. This fact means that any natural language generator using plans as input must customize itself somewhat to the peculiarities of the particular planner it is working with. One problem peculiar to NONLIN's representation language concerns the manner in which preconditions are specified. In NONLIN an operator specifies how a goal is expanded to a network of subgoals. As was observed in the earliest AI planners, the most common case is that a goal has preconditions, goals that must be true before the given goal can be achieved. In NONLIN, one has to use an expansion to represent this, with the consequence that one wants the original goal itself to occur in the expansion (that is Goal expands to Pre^1, . . ., Pre^N, Goal). NONLIN will not allow this, so there have to be two distinct representations of the goal. So in the car example, we have {goal . . .} for the high level goal and {act . . .} for the low level version (although this scheme might not work in every domain). By this means we can make NONLIN behave, but give ourselves a linguistic problem—every action occurs twice. For example, we might get:

STARTING THE ENGINE
Go to the cab and then start the engine and you will have finished starting it.

Roughly speaking, the distinction is between 'starting the engine' (the whole task) and 'actually starting the engine' (the specific operation). To some extent we can avoid the problem by using different phrases (e.g., 'turn on the engine'), but it does not make the generation task easier.

5.3 DEFICIENCIES IN OUR APPROACH

The problems with our natural language accounts are, of course, not entirely due to deficiencies in the plans we are working on. We have deliberately held closely to some basic guiding principles to evaluate their applicability. So it is important to pinpoint their failings in our current system and mention possible alternative approaches.

RELYING ON PLAN STRUCTURE

To build a domain-independent system to generate text from plans, we have deliberately tried to use only information that the planner itself understands; i.e., information about the structure of the plan. One of the

fundamental tenets of our approach was thus that the plan abstraction hierarchy would be a useful source of information about how the text should be organized. But our experience suggests that it may not be as useful as one might think. As well as the kind of problems described above (which might be corrected in a different planning system), there seem to be more general discrepancies between the kind of abstraction useful to a planner and the kind useful to a text generator.

For example, our car domain and many of the blocksworld plans that have been studied in AI tend to have a deeper abstraction hierarchy than one might expect from the apparent simplicity of the tasks. A generator that tries to exploit them all ends up producing too much structure in its text. Thus the example used in Section 3 also has a 'section':

GAINING ACCESS TO THE DISTRIBUTOR
Detach the dirt cover from the engine and you will have finished gaining access to the distributor.

Here there is a level of abstraction that is useful to the planner, but not to the human reader: it would probably have been better to insert this section "in-line" in the higher level description. On the other hand, the single "section" devoted to installing the services in the house-building plan would have gained from being broken up in some way. There may be a linguistic solution to the problem of whether a piece of information deserves a full "section," perhaps in terms of a domain-dependent model of what is and is not worth saying, or the problem may point to a fundamental difference between the ways the planner and a human perceives the planning task. Either way, what is clear is that the planner's abstraction hierachy alone is not fully adequate for text generation. Whether we can devise general principles for producing alternative decompositions of plans more suitable for text generation remains an open research area.

REPETITION

We have commented above on reasons why the raw material we can gain from plans is liable to lead to repetitiveness in the text. Even if we managed to enrich the plan representations suitably, however, the generator would still be deficient when the input really is uniform. In particular, the uniformity of the text output often leads to unwanted ambiguities, simply because of the lack of variation in the stylistic devices used. For instance, in the following excerpt it is unclear whether the potato peeling is supposed to be "in parallel with" melting the fat, or just with stirring the sauce:

Melt the fat, . . .
After this, stir the sauce until it boils.
Meanwhile peel the potatoes and cut them into pieces.

We originally hoped to overcome the problem of repetition by providing several structure-building rules for each type of message language construction, which would be sensitive to the form of the objects involved in the construction. To some extent we succeeded in producing such rules, but the effect on the text was not great. The problem here is that, even with these extra rules, our structure-building is based solely on *local* patterns in the message, whereas the problem of repetition can only be solved by a *global* planning of the text. It might be possible to gain some improvement in our system by having the choice of structure-building rules be determined partially by some random factor, but a proper solution requires a more radical redesign.

LINGUISTIC SIMPLIFICATIONS

There are a number of stylistic issues that the system cannot easily accommodate. For instance, operations such as "heavy NP shift," segmentation into sentences, coordination, and ellipsis all require detailed stylistic control and evaluation. The message language is deliberately nonlinguistic and so can only approximately represent the kinds of language-dependent stylistic information such processing needs. For instance, rewrite rules can decide how to group information on the basis of the complexity of the *message*, but this only indirectly reflects the complexity of the *text* that will be generated. The effective use of different stylistic devices depends in the end on simplifications that are justified on linguistic, rather than conceptual, grounds, and this suggests that our architecture should really incorporate a style module capable of reasoning at this level. Such a style module would necessarily have to take a more global view, looking at the overall linguistic effect of the localized basic text generation processes. It might be possible to introduce linguistic simplifications at structure-building time, relaxing the requirement of compositionality (indeed, this is how McDonald 1983 operates). We believe, however, that it would be preferable to attempt to treat it at least conceptually as a subsequent processing stage.

5.4 WAYS FORWARD

In this paper we have described a system for generating natural language from automatically generated plans. Our main aim in developing the system was to produce a model of a complete system using state-of-the-art methodology and techniques, partly to evaluate the current state of knowledge, and partly to provide a basis for comparison for future work. Logically, then, there are two strands to further work based on this research: building on the evaluation to learn lessons about the design of generation systems and the systems they interact with, and building on the system itself to produce better generation-from-plans systems.

One of the key evaluative lessons concerns the plan structures a system like this depends on. We found the plans produced by NONLIN unsatisfactory and we have begun to understand why. We must now specify what we would like a plan to look like and contain, within the general constraint that a planning system could reasonably produce it. Our present approach to

this topic is to take a very formal view of plans as algebraic expressions over states (rather than actions or goals) with a well-defined formal semantics, allowing us to be clear about the semantic effect of plan transformations.

The system itself falls broadly into two parts, building and simplifying the message, and turning the message into text. Of these the latter is the more modular, more declarative, and probably more successful at present. To a certain extent it can serve as a piece of enabling technology for research on the message component. Its major deficiency as discussed above is global stylistic control. Its handling of morphology is currently rather unprincipled, but the utilization of a morphological representation language such as Datr (Evans and Gazdar 1989 a,b) would rectify this.

The biggest outstanding task, however, is the message planner itself. The mechanism described above employs some quite powerful techniques in a fairly effective way, but it is not very perspicuous or extensible. We have begun work on a new message planner module that applies transformation rules to plans of the algebraic type mentioned above, gradually transforming the plan into an optimized message structure. This will provide us with a rule-based semideclarative framework in which to explore further the issues of message planning.

Acknowledgments

The work reported here was made possible by SERC grant GR/D/08876. Both authors are currently supported by SERC Advanced Fellowships.

References

Chester, Daniel 1976 The Translation of Formal Proofs into English. *Artificial Intelligence*, Vol 7, 261.

Conklin, E. Jeffrey and McDonald, David D. 1982 Salience: The Key to the Selection Problem in Natural Language Generation. In: Proceedings of the 20th Meeting of the Association for Computational Linguistics, Toronto, Canada.

Dale, Robert 1986 The Pronominalization Decision in Language Generation. Dissertation Abstracts International Report 276, University of Edinburgh, Edinburgh, U.K.

Evans, Roger and Gazdar, Gerald 1989 Inference in Datr. In: Proceedings of the 1989 European Association for Computational Linguistics, UMIST.

Evans, Roger and Gazdar, Gerald 1989 The Semantics of Datr. In: Proceedings of the 1989 Artificial Intelligence Society of Great Britain, University of Sussex.

Gazdar, Gerald; Klein, Ewan; Pullum, Geoffrey; and Sag, Ivan 1985 *Generalised Phrase Structure Grammar*. Blackwell.

Grosz, Barbara J. 1977 The Representation and Use of Focus in Dialogue Understanding. SRI Technical Report 151.

Kay, Martin 1979 Functional Grammar. In: Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society. Berkeley, CA.

Kay, Martin 1984 Functional Unification Grammar: A Formalism for Machine Translation. In: Proceedings of COLING-84. Stanford, CA.

Mann, William C. and Moore, James A. 1981 Computer Generation of Multiparagraph English Text. American Journal of Computational Linguistics, Vol 7, No 1.

Mann, William; Bates, Madeleine; Grosz, Barbara; McDonald, David; McKeown, Kathleen; and Swartout, William 1982 Text Generation. American Journal of Computational Linguistics. Vol 8, No 2.

McDonald, David D. 1983 Natural Language Generation as a Computational Problem: An Introduction. In: Brady, Michael and Berwick, Robert (Eds.), *Computational Models of Discourse*, MIT Press, Cambridge, MA.

McKeown, Kathleen R. 1982 Generating Natural Language Text in Response to Questions about Database Structure. Ph.D. thesis, University of Pennsylvania, Philadelphia, PA.

Sacerdoti, Earl D. 1973 Planning in a Hierarchy of Abstraction Spaces. In: Proceedings of the Eighth International Joint Conference on Artificial Intelligence-83. Palo Alto, CA.

Sacerdoti, Earl D. 1975 The Non-Linear Nature of Plans. In: Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, USSR.

Sidner, Candace L. 1979 Towards a Computational Theory of Definite Anaphora Comprehension in English Discourse. Technical Report 537, MIT Artificial Intelligence Laboratory, Cambridge, MA.

Swartout, William R. 1983 XPLAIN: A System for Creating and Explaining Expert Consulting Programs. *Artificial Intelligence*, Vol 21.

Tate, Austin 1976 Project Planning using a Hierarchical Non-linear Planner. Dissertation Abstracts International Report 25, University of Edinburgh, Edinburgh, U.K.

Tsang, Edward 1986 Plan Generation in a Temporal Frame. In: Proceedings of the 7th European Conference on Artificial Intelligence, Brighton, U.K.

Notes

1. Current address: Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, EDINBURGH EH1 1HN, United Kingdom.

2. In the node descriptions we distinguish explicitly between goals and actions that achieve goals. The reason for this is discussed in Section 5 below.

3. It is possible, however, to specify that, for a given domain, there will only ever be one agent.