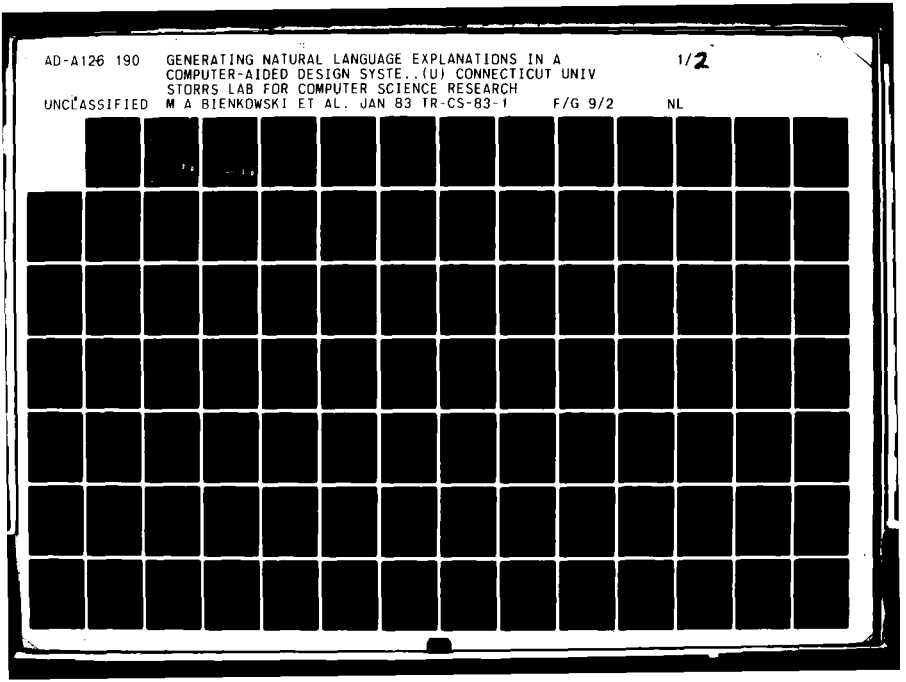
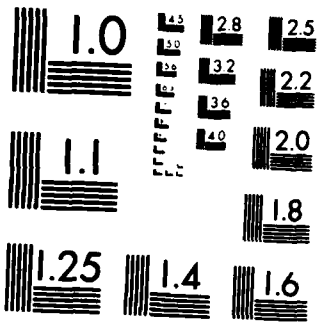
AD-A126 190 GENERATING NATURAL LANGUAGE EXPLANATIONS IN A COMPUTER-AIDED DESIGN SYSTE..(U) CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RESEARCH

1/2

UNCLASSIFIED M A BIENKOWSKI ET AL. JAN 83 TR-CS-83-1 F/G 9/2 NL





COMPUTER SCIENCE TECHNICAL REPORT

Laboratory for Computer Science Research
The University of Connecticut

COMPUTER SCIENCE DIVISION

Electrical Engineering and Computer Science Department
U-157
The University of Connecticut
Storrs, Connecticut 06268

83 03 10 049

(8)

GENERATING NATURAL LANGUAGE EXPLANATIONS

BY

Marie Bienkowski

Technical Report CS83-1

JAN 1983

Generating Natural Language Explanations in a Computer-Aided Design System

M. A. Bienkowski, R. E. Cullingford and M. W. Krueger

Department of Electrical Engineering and Computer Science
The University of Connecticut
Storrs, CT 06268

ABSTRACT

CADHELP is a graphics-based computer-aided design system which contains detailed knowledge bases intended to support three different types of "intelligent" behavior: (1) the generation of natural-language explanations concerning the operation of the graphical features, for use by naive users [CULL81, CULL82]; (2) an animated display [NEIM82], coordinated with the explanation, simulating the feature's use; and (3) control of the operation of the CAD system itself, by interpretation of knowledge structures describing the system's operation, [KRUE82]. This final report is a detailed description of the knowledge-base summarization and generation methods developed for CADHELP, which are the basis for the three different sources of knowledge.

The Explanation Mechanism devised for CADHELP describes the CAD commands CADHELP can execute using text and graphical animation. A unique feature of this system is that neither the text nor the animation are stored, but are generated from a representation of knowledge about how to use CADHELP. This representation, called a feature script, is a set of concepts linked by causal relations. Since the feature scripts developed were very detailed to enable the animator to work, a means of pruning the script to produce natural-sounding text was needed. A selector mechanism, called HELPCON, was developed to select concepts for expression from the feature scripts using rules on how to conduct an explanation. The concepts thus selected are generated as English sentences by another module called CGEN, which prunes a concept to express it in a concise form much as HELPCON does.

To test the explanation methods and generation strategies developed for CADHELP, a design was produced for an Explanation system for the Academic Counseling Experiment currently under development. While the language generator, CGEN, was moved to this new domain with ease, a new Explainer was needed. This suggested a general model for the development of such language producing systems; a domain-independent language generator which interfaces with a domain-dependent concept selection mechanism.

This research was supported by Navy Contract N00014-79-C-0976.

Selected Project Bibliography

[CULL81]
Cullingford, R.E., Krueger, M.W., Selfridge, M.G, and Bienkowski, M.A. 1981. Towards automating explanations. Proc. 7th Int. Joint Conf. on Artificial Intelligence. Vancouver, B.C., Canada. (August)

[CULL82]
Cullingford, R.E., Krueger, M.W., Selfridge, M.G., and Bienkowski, M.A. 1982. Automated explanations as a component of a computer-aided design system. IEEE Trans. SM&C. Special issue on human factors and user assistance in CAD. Vol. SMC-12, No. 2, Mar/Apr 1982.

[KRUE81]
Krueger, M.W., Cullingford, R.E., and Bellavance, D.A. 1981. Control issues in a multimodule CAD system containing expert knowledge. Proc. 1981 IEEE Int. Conf. on Cybernetics and Society. Atlanta, GA. (October)

[NEIM82]
Neiman, D. 1982. Graphical Animation from a Knowledge Base. Proc. 1982 Conf. of the American Artificial Intelligence Society. Pittsburgh, PA. (August).

Table of Contents

List of Figures

CHAPTER 1

Introduction

1.1 Generation of Language

Generation of natural language by computer is becoming increasingly important. As computers are used more and more to perform tasks requiring that they interact with naive users (e.g. as providers of information on phone numbers, airline reservations, etc) it is necessary that they be able to produce language in a natural and flexible way. In fact, the most friendly computer interfaces [Haye79], will have analyzing and generating programs whose language behavior mimics that of human beings. For a generation program to model human language performance, it must be capable of more than mere translation of some input representation of a sentence into English. Such a program must be able to determine what to say in order to communicate with a user in a natural fashion. Part of this naturalness comes from knowing what not to say. If a set of knowledge, represented in its entirety in a program's memory system, is to be communicated, only part of it need actually be said. Some of it can be inferred by the listener or is not important to the communication. In order to build programs that can make the distinction between useful and useless information, it is essential to have an understanding of the techniques people use to decide what (and what not) to say.

Language generation tasks can be organized hierarchically according to their complexity. Generation of a single sentence with little surrounding context is simple, but by no means easy. For example, sentence 1 is understandable to people without any additional information, even if the person Jacob is not known, and thus should be easy for a computer program to generate.

1) My friend Jacob was looking for his shoes yesterday.

There are difficulties, however. The first problem is representing, in computer-usable form, the complicated ideas expressed, namely that some social relationship exists between two people which is used to identify the actor of an action, and that this actor did some time in the past perform the action of looking for shoes that belonged to him. Once these ideas are represented in some form a program can manipulate, this internal representation must be translated into a linear sequence of words. The complex notion of possession and use of shoes is translated into "his shoes" as opposed to "the shoes belonging to and often worn by Jacob." The temporal relationship between the time of the action and the present time is condensed into the tensed form of look, i.e. "was looking," and the concept "a person named Jacob who has the social relationship of friendship with me" becomes "My friend Jacob." An important part of the study of language generation is investigations into how these shorthand forms are produced. It will be necessary to model these in any program which is to generate natural sounding output.

The next level of complexity in generation can be found in the production of paragraph length text. Here the individual sentences

must be connected using conventions that tie the sentences together to form a unit. For example, the sentences given below represent a cohesive unit.

2) My friend Jacob went to buy some cooking apples today.
3) Some of the apples he bought were bad.
4) I told him to take the bad ones back.

Sentences 2 to 4 illustrate some common phenomena occurring in paragraph length text. Note the different ways of referring to the apples. They first appear in sentence 2 as "some cooking apples." Next, in sentence 3, the apples are referred to as "the apples he bought," ignoring what type of apples they are, and informing us that the apples were in fact obtained. A new item, the group of bad apples, is introduced in 3 as a subset of all the apples bought. Finally, in sentence 4, no explicit mention of the apples bought is made, they are only implicit in the mention of those that were bad in "the bad ones." It is also interesting to observe what has been left out in each sentence, for example, the mention of the store where Jacob went and bought the apples. This omission might not have occurred if the intent of the story was to tell someone that some particular market sells bad fruit. This example illustrates that in more complex generation tasks what to say and how to say it are functions of what has been said before, what the listener can be expected to infer, and what the intent of the conversation is.

From a knowledge engineering viewpoint, generation of language in a mixed-initiative conversation is a difficult task for computer programs (e.g. [Reic78], [Carb70]). Mixed-initiative conversation, where

either participant can select the topic of discussion, has complications well beyond that of generation of single sentences or paragraph length text. One obvious problem is having to analyze the input from the other participant for meaning. Not only does the computer as a conversationalist have to understand what a speaker is saying, it must know how to use that understanding to determine what the speaker knows about a subject. In a text generation task a program can expect that the user knows what he has been told, and what he has deduced from what he has been told. In a conversational system, however, the problem of deciding what to say is more complex since more information about what the user knows is available. For example, an appropriate answer to question 5,

5) Who has been eating the eclairs?

might be the ellipsed response,

6) John has.

Sentences 5 and 6 illustrate that deciding how much can safely be deleted from an utterance in a conversation depends upon the preceding context (i.e. what the user and program say) and how much the program can assume the user knows about the topic of discussion. It shall be argued that imposing limitations on these types of knowledge can lead to a program which is feasible to implement.

1.2 The Study of Explanations

For purposes of study, the generation process as described above can be divided into two subprocesses, one which decides what concepts to express from a memory structure (e.g. a database), and the other

which actually says it in a human language (as in [McDo81], [McKe80]). Decisions regarding what to say, however, occur throughout the generation process, from the time it is decided that something is to be selected from memory for expression (perhaps requested by a user or another program) until it is actually expressed in the words of the language. For this reason, it is most informative to study language generation by examining problems in both subprocesses. A useful vehicle for this exploration is the study of explanations, for several reasons. First, explanations can be given as descriptive paragraphs, and an implementation can be produced without the distracting details associated with conversational interaction. Such a system, however, could be expanded to accept input from the user regarding the ongoing explanation. This limited conversational ability could be used to clarify or reexplain any unclear ideas. Secondly, restricting the task to explanation defines exactly what the system and user can be expected to know, i.e. the system can be expected to know a lot, and the user not too much. This leads to a specification of the pragmatic information and world knowledge such a program should have regarding the contents of the user's head. Third, if the domain to be explained is well-defined, a situational context can be established and used by the program to assist in generating natural sounding output. The situational context refers to the physical situation experienced by the user, and includes things such as the tools commonly used for a task and actions associated with those tools (which may need to be initially explained, but then become part of the context). An example of this use of situational information in explanations is the use of

specialized terms, e.g. in giving a recipe, one may state

> Butter and flour a 9-inch cake pan.
> To butter and flour a pan, rub butter on it then sprinkle flour on it and shake out any excess flour.

After this short explanation, the verbs butter and flour can be used without further explanation.

There are many types of explanations that occur but they can be distinguished by the kind of information being explained. One class deals with describing to someone how a physical process works, a process that does not necessarily involve them or another as an actor (except perhaps to initiate the process). This class would include things like an explanation of a biological mechanism or a car's ignition system. The explanation of such a system is straightforward to program, assuming no interferences occur in the system (as in [Rieg77]). A second class of explanations deals with relating to a potential actor how to act to affect the physical world in a desired way. An example of this is how to tune-up a car. Here the explanation is more complicated since such a system must have knowledge of what the user knows, and must be able to represent all the things a naive mechanic could do *wrong*. A final class are those explanations that describe the complex interaction of an actor (or actors) with social institutions. Examples of this category include events like how one opens and uses a checking account, where physical acts may be involved (e.g. one has to drive to the bank and sign some papers) but the main thrust is the complicated notion of banking.

The class of explanations described above do form a hierarchy, i.e. solving the first can lead to a solution of the second and so on. One of the reasons the solutions build upon another has to do with the representation of the ideas. Once the laws of physical causality can be represented, the notion of actors influencing this causality can be also. Representations of actors interacting in social situations, with underlying physical laws, then follow. Representation of meaning in computer-usable form is difficult, and many representational systems have been proposed ([Mins75], [Scha75], and see [Barr81] for an overview). As an example of the complexity involved, consider the examples suggested above. A car's ignition system can be represented by a sequence of events, each one enabling or causing another (as in [Rieg77]). The representation of how to tune-up a car is more complicated, since the behavior of the mechanic must be represented, as well as the result of that behavior on the car. More complex still is representing how to open a checking account. Ideas such as money, and a bank's holding onto your money then paying it out to certain other people and institutions that you present signed pieces of paper to, are not easily captured in a form a computer understands.

Once a representation is decided upon some mechanism for selecting concepts from the entire set of knowledge is needed. There are many reasons for representing the knowledge in its entirety, and for representing it in an abstract form. The strongest reason is that the same knowledge may be used for a number of tasks: language generation, story understanding, planning, question answering, etc. If the system generating explanations will know more than it should say, a selector

mechanism which embodies a method of examining a piece of knowledge and selecting from it those concepts to be expressed is needed. The selector uses rules that are dependent upon several things. One is the domain being explained. For example, the naive mechanic must be explicitly told every step in timing a car, but the person new to the banking world is helped along by persons operating in their functions in the bank and need not be told all the details. Another factor influencing the selector rules is the intent of the explanation, i.e. "Why does the listener want to know?" If the listener is actually going to attempt a tune-up, more detail will be given (including precautions: "Don't get to near the fan when the car's running"). A third factor used in the rules is general knowledge of how to conduct an explanation. These include ideas like: "If you use a potentially unfamiliar term, define it," or: "Say things in the order in which they occur."

The pieces of the total set of knowledge chosen by the selector must eventually be translated into a language. A good characteristic of a language generator is that it be task-independent, and have any domain dependencies that did exist well specified (see [McDo81]). This would allow the generator to apply to almost any domain with minor alterations. Another feature desirable for a language generator is the ability to perform paraphrase [Gold75]. In addition, it should be easy to model any observed phenomena of language with a generator, such as generation of passives or imperatives.

Using the hierarchy of explanations described above, and keeping in mind that a computer implementation of an explanation mechanism is desired, the domains to use for explanations can be selected. The areas used in this research, and the reasons for their use, are discussed in the next section.

1.3 Domains for Explanations

The domain chosen for a study of the explanation process has a great influence on the complexity of the system developed. Previous work on explanations has arisen from the necessity to have computer programs explain their behavior to users or programmers. The MYCIN rule-based system [Shor76] explains its conclusions by describing the chains of rules that have fired, including why a rule did or didn't fire (the test part) and what conclusion was reached (the action part of a fired rule). MYCIN contained no explicit attempt to model the generation ability of an expert, it was more of a convenience for the developer and user. Swartout [Swar77] developed a system to explain the actions and procedures of programs written in OWL to provide digitalis therapy. Like MYCIN, Swartout's system explains methods that are used by the advisor to reach a conclusion, but can also explain methods that could potentially be used. In contrast to the explanations of code-like knowledge done by Swartout and Shortliffe, McKeown [McKe80] outlines several principles that could be applied in explaining the contents of a database which is a more static structure. For example, methods such as comparing and contrasting items, use of analogy and illustration through example assist in providing a better

description. McKeown's work on the description of static information stored in a database is more in line with the current work.

Systems that frequently need explaining are computer programs designed to assist a user in accomplishing a task (e.g. operating systems, editors, etc.). These can conveniently be described as falling within the second level of the hierarchy described in the previous section, namely, an actor interacting with the physical world. In choosing a computer system to explain, a desirable one is a system that offers sufficiently challenging problems in generation of English, yet is not so complicated that the knowledge an expert has of the system is too complex to represent. A computer-aided design (CAD) domain has both these features. There is a finite set of commands such a system can perform which can be explained. There is also a definite physical environment experienced by the user, e.g. graphics screen, terminal, input devices, which can be represented, and used for situational context. It is also relatively easy to model the overt actions of the user and the system during the design process in terms of simple actions.

There are three types of explanations that can be built into a CAD system. One is the simple explanation of how to use a command. This is performed at the request of the user and describes in varying levels of detail how the command is actually performed. Another kind of explanation is prompting text which guides the user through a complicated feature by reminding him of his expected behavior, or notifying him of the occurrence of events of interest. A third type is a

HELP facility, which rescues the user who has made a mistake, and attempts to describe to him what went wrong.

In most CAD systems [Marc82],[Fenc82], the explanations and prompts are simply stored text. This becomes tedious for experienced users, and makes the programming of a HELP facility difficult. If the text is not stored, but generated from some stored representation, the explanations given can become more and more laconic as the user gains experience. In addition, if the system knows how the feature is supposed to be executed, when errors occur, it can determine what they were in a flexible manner. An additional benefit of having the system actually know how the features operate is to have the explanation of a feature occur in a modality other than language, for example, to produce graphical animation to assist in the explanations.

Considerations such as these led to choosing to develop an explanation program for the use of a system called CADHELP for this research. Explaining the system consists of describing how each feature is to be performed (e.g. adding a gate to the design or connecting two gates). Each command the CAD system can execute is stored in a knowledge structure called a feature script. The notion of script, a stereotyped sequence of actions, is taken from [Cull81a]. Each feature script is a detailed description of the expected behavior of the user and the response of the system to that behavior during the execution of a particular command. These feature scripts are detailed enough to be used as the input for generation of English and the generation of graphical animation [Neim82]. The exact nature of the

representation and the processes that operate on it to produce text are discussed in Chapter 3.

Explanations of the use of a CAD Tool suffer from several shortcomings when used to model computer generation of language. One is the problem of extensibility, i.e. once all the features have been explained, only adding additional features produces new generation tasks and the problems are similar to those experienced in explaining the other features. Additionally, the CAD explanation system was not designed to be interactive, (unlike the actual computer-aided design part), and extending it to be so would be a laborious task. For reasons such as these it was decided to explore the explanation process in a different domain. A system called ACE (Academic Counseling *Experiment*), *currently under development at the University of Connecticut*, was chosen for study.

ACE models an academic counselor who performs various tasks for a student such as conducting a preregistration or answering questions about courses in a mixed-initiative fashion. The task of the explainer is to describe to a new undergraduate how one goes about obtaining a degree in Computer Science at the University of Connecticut. This type of explanation deals with an actor interacting with a complex social organization (the University) to accomplish some goal (lifetime happiness beginning at 22K). This new domain is different enough from the CAD domain to present new problems in both language generation and concept selection.

1.4 Outline of the Thesis

Chapter 2 presents a model (developed by [Cull81b]) for an essential part of any explanation system, a generator of English sentences, which was extended for use in this work. After discussing previous work on generation, the important topic of the representation of the meaning of the sentences to be generated is covered. The notation of Conceptual Dependency is shown to be a suitable one for the purpose, and examples of its use in representing input as well as word meanings are given. The actual generation process, implemented in a LISP program called CGEN, is then described in two parts. The first deals with the underlying control cycle that produces a linear sequence of words from a Conceptual Dependency (CD) representation. The second covers the generation strategies that operate on concepts to assist in the production of natural sounding utterances.

Chapter 3 discusses the development and implementation of an explanation subsystem for the CAD domain described above. To provide a framework for the discussions on the explanation strategies used in this domain, as well as making the examples more understandable, the actual CAD system is explained. Next, representation is again discussed, this time for more complex, structured knowledge. The purpose of these more complex structures is to represent in the computer the knowledge of the CAD tool an expert has. The knowledge structures used for the CAD domain were influenced by the decision to generate English from a CD representation, and are based on CD representation theory. The discussion then centers on the explanation model for this domain

as embodied in a LISP program called HELPCON. HELPCON examines the knowledge structures representing the features of the CAD Tool, and selects certain concepts for expression by CGEN. The chapter ends with an excerpt from an example of the execution of HELPCON.

The next chapter, Chapter 4, describes a design for a program capable of producing explanations in a more complicated setting, namely the academic counseling project described above. The only parts of this design actually implemented so far are the generation of some key sentences characteristic of the ACE domain. The chapter begins with an outline of some extensions that would be useful for a next-generation explainer, based on experience with the explainer for the CAD domain. Next, an overview of the ACE system is given. The exposition following this describes the strategies designed for use with ACE. The last section then gives a hypothetical example of the run of this new explanation system. This is followed by a summary and discussion of extensions in Chapter 5.

CHAPTER 2

Conceptual Generation

2.1 Previous Work in Generation

The question of generating sentences from some representation of their meaning has been the subject of research in artificial intelligence and computational linguistics, and several language generators exist which are quite capable. Winograd's [Wino72] SHRDLU used canned phrases, template sentences and a noun phrase generation algorithm for generating output, and also had a set of dialogue heuristics for increasing the naturalness of its responses. Winograd's system for generation, however, is specific to the blocks world application for which it was designed. A generator that was also done for a limited domain was Chester's [Ches76] EXPOUND. EXPOUND is a system for expressing predicate calculus proofs in English. Its main focus is the structuring of the lines of a proof into an English paragraph, and so has a simplified generator for the actual sentences. Basically, each logical predicate EXPOUND knows simply has an associated verb and function words that connect the arguments to the predicate.

Simmons and Slocum's work [Simm72], was an early approach to a more general theory of generation. The input concept to their generator was in the form of a phrase marker. This input was passed to a phrase structure grammar (represented using an ATN) which supplied an ordering for the semantic components of the concept. Simmons and

Slocum's work was influential for later research on conceptual generation. In particular, Shapiro's [Shap75] work on generation of English was an attempt to extend their work to provide a generator with the ability to determine the value of certain attributes that Simmons and Slocum's generator took as given (e.g. tense and modal specifications).

Davey [Dave74], like Winograd, adapted Halliday's systemic functional grammar but for use in a generation program rather than a parser. Davey's program was one of the first to attempt to model the speaker's use of language (in a specific context) as a communicative device. He used the simple domain of describing the moves of a tic-tac-toe game and was able to provide some rationalizations for the use of particular linguistic forms. For example, the rule for connecting moves with a coordinate conjunction would only apply if the aspects of the moves being connected were equal. Unlike previous generators, Davey's program used domain specific knowledge to justify its use of the language.

McDonald [McDo81], in his work on MUMBLE, makes a distinction between the speaker component of a generation program (decides what to say) and the linguistic component (decides how to say it). His argument for making this division is that it frees the researcher using the linguistic component, (which MUMBLE represents), from having to worry about a specific input representation. For any domain, an interface program and dictionary is built to provide the necessary translations. The interface program translates a message from the speaker's

internal representation into a surface structure representation, which is then linearized into a sentence. He has tested his linguistic component using six different speaker programs, including the domain of Chester.

Goldman's model of language generation [Gold75], embodied in the program BABEL, departs from other theories of generation by beginning with a conceptual representation of the ideas to be expressed (in Conceptual Dependency format) and producing an English sentence for it. BABEL selects a verb sense for a concept by consulting a discrimination tree attached to the concept's primitive action or state. The verb sense is represented as a syntax network, modeled after Simmons and Slocum's *marker nets*. BABEL fills in the cases of the syntax net using the conceptual cases of the input concept. The resulting net is then linearized into an English sentence using an ATN (again following Simmons and Slocum). Salient features of BABEL are, 1) its strength in performing paraphrase, 2) its ability to make subtle distinctions between words with similar meaning, and 3) its generation of German as well as English from some input concept. BABEL is able to perform all these tasks since it has some notion of what a word means, unlike other generation systems. A generator that was built from BABEL was used in Meehan's TALE-SPIN [Meeh81], a program which uses the planning structures of the Schank and Abelson theory [Scha77] to produce simple stories. Meehan makes a strong case for needing constant references to a memory system for producing coherent text, for example, using pronominalization and conjunctions. The success of BABEL and TALE-SPIN in generating from a conceptual representation prompted further

investigations into language generation using this input. In particular, Cullingford et. al. [Cull81b] designed a conceptual generator for use in several systems. The design and function of this generator, as well as additions made to it for the current work, is described below.

2.2 Representation

An important issue for any generation system is what the representation of the input will be. Unlike parsing systems, where the input is sentences of the language, the input for a generator can be anything from a phrase marker to a predicate calculus formula. There were several requirements that motivated the choice of representations for CGEN, the conceptual generator developed by Cullingford et. al. One was that, since it was to be initially used in a computer-aided design domain, other programs would need to manipulate the representation besides the language generating program. In particular, a system was under development for producing graphical animation from the representation, so some notion of what a word means at a basic level was important. Paraphrasing is also a task that was desirable for the generator to be able to perform, and if a representation is unable to capture the underlying similarities among the meaning of words, a program using it cannot perform paraphrasing [McDo81]. In addition, it appears that the decision of what word to use in expressing a concept is dependent upon the meaning of the other concepts that are associated with it. For example, the distinction between run and walk depends upon the quickness of the step used, but both are forms of movement. Also, use of an abstract representation of meaning may pro-

vide a motivation for the use of linguistic phenomena such as the passive or relative clauses.

Given these requirements, some theory that used the notion of semantic primitives was needed. Wilk's system for machine translation [Wilk76] represents word meanings as sets of descriptive semantic features that describe the class the word belongs to as well as its distinguishing characteristics. However, Wilk's semantic formulas for word-sense meaning are not of sufficient generality to be useful for several programs, all of which need the knowledge for different purposes, to use. Norman and Rummelhart [Norm75] also developed a semantic primitives based representation but were more concerned with the psychological reality of their primitives than an actual computer implementation.

For Cullingford et. al., the theory behind Conceptual Dependency [Scha75] met the requirements for an input representation for CGEN. One basic premise of CD theory is that a representation of meaning should be language free, and should explicate the relationship between utterances that are close in meaning, but may have different surface forms. So, an analyzer parsing into CD format should produce very similar CD representations for sentences 6 and 7, and a generator should be able to produce them as paraphrase.

6) I got an A in CS110.
7) I took CS110 and got an A.

Representing the knowledge of a system in a CD format also enables generation of output in modalities other than language.

Any concept represented in CD format contains a conceptual class which identifies the underlying action or state the concept expresses, if the concept is a full conceptualization, or identifies the primitive type of the concept, e.g. person, polity. The rest of the concept is a set of slots and their associated fillers. Every conceptual class has a unique set of slots, and these slots and their fillers serve to convey information about the primitive action or state. For example, one conceptual class is PTRANS (Physical TRANSfer). In order to describe an event which is a PTRANS, the ACTOR, OBJECT, TO, FROM, and INSTRUMENT slots may be filled. For example, a simple CD representation of the sentence: "John went to Hartford" is shown in Figure 2.1. Further discussion of CD representation can be found in [Scha75].

Deciding to use Conceptual Dependency notation to represent the input concept to be generated determines, in part, what a dictionary entry for a word must look like. The base meaning given to a word in

---

```
(PTRANS ACTOR (PERSON PERSNAME (john)
                      SURNAME (nil) GENDER (masc))
        OBJECT (PERSON PERSNAME (john)
                       SURNAME (nil) GENDER (masc))
        TO (POLITY POLNAME (Hartford) POLTYPE (city))
        FROM (nil)
        INSTRUMENT (nil)
        TIME (TIMES TIME1 (past) TIME2 (nil))
```

The Representation for "John went to Hartford."

Figure 2.1

---

the dictionary is a CD frame, a concept with some of the slots empty (nil) and some of them filled (see Figure 2.2). The empty slots can match anything in an input concept, but the filled slots, which represent restrictions on the slot fillers, must match exactly. For example, the filler of the ACTOR slot in the definition above is restricted to be the system, whose name is CADHELP. Some restrictions on

```
~ DELETE
{makdef del1
          ~ word for this definition
        (delete)
          ~ concept frame representing the meaning
          ~ of the word 'delete'
        (MTRANS ACTOR (PERSON PERSNAME (cadhelp) ROLE (*sys))
                MOBJ (nil) INST (nil) FOCUS (nil)
                MODE (nil) FROM (nil) TO (nil))

          ~ active syntatic predicates for the word delete
      ({(actor) [((ACTOR)    (pr parent))
                 ((MOBJ)     (fo parent) (pr (path FROM)))
                 ((FROM)     (fo parent) (fo (path MOBJ))
                             (fo (fw from)))]}

          ~ passive syntatic predicates for the word delete
       {(mobj)  [((MOBJ)     (pr parent)
                 ((FROM)     (fo parent) (pr (path ACTOR))
                             (fo (fw from)))
                 ((ACTOR)    (fo parent) (fo (path FROM))
                             (fo (fw by)))]})

          ~ semantic predicates expressing restrictions
          ~ on the fillers
        [(eq (conclass (grv '(MOBJ) :input-concept)) '#device)
         (equal (grv '(FROM ROLE) :input-concept) '(*design))
         (equal (grv '(TO) :input-concept) '(nil))]
}
```

CGEN Dictionary Definition of the Word 'delete'

Figure 2.2

slot-fillers cannot be represented by a simple pattern, for example, if the restriction is that a filler must belong to a class of items, or that two fillers must be equal. In this case, a semantic predicate is used. Semantic predicates are conditions on slot fillers expressed as LISP expressions. For example, the word delete in CADHELP is only used if the object being manipulated belongs to the class of devices, i.e. delete-able objects, and if the manipulation on that device is a transfer from the design to nil (nowhere). This notion is embodied in the LISP code shown in the final three lines of Figure 2.2.

The final component in a dictionary definition is the specification of syntax for sequencing words. Recall that Goldman's BABEL used a syntax net associated with a main verb which was run through an ATN to linearize the concepts. This extra processing appears unnatural and carries unnecessary overhead. What is needed, at a most basic level, is the notion that one word or concept precedes or follows another. This is precisely the kind of syntatic specification that CGEN uses. Most words in the dictionary, (e.g. delete from Figure 2.2), are associated with a set of precedes and follows predicates. If the word found is a main verb, then depending upon the value of the FOCUS role in the input concept, syntatic predicates that produce either an active or a passive sentence are chosen. The predicates state where the parts of the concept that CGEN is trying to express are to be positioned relative to three things, 1) the word found, 2) the other concepts to be expressed, and 3) function words that serve to mark the filler of a particular slot. (Function words are connective words that are associated with a particular action word and a particular slot.)

For example, an English summary of the syntatic predicates shown in Figure 2.2 might be as follows:

If the focus is on the ACTOR, then:

(1) Say the concept expressing the ACTOR before the parent word, delete.

(2) Say the concept expressing the MOBJ (mental object) following the parent word delete, and before the FROM slot.

(3) Say the concept expressing the FROM slot following the parent word delete, following the expression of the MOBJ filler, and following the function word from, (an actual lexical item to be said).

If the focus is on the MOBJ, then:

(1) Say the concept expressing the MOBJ before the parent word, delete.

(2) Say the concept expressing the FROM slot following the parent word delete, before the ACTOR slot filler and following the function word from.

(3) Say the concept expressing the ACTOR following the parent word delete, following the expression of the FROM filler and following the function word by.

The input to CGEN, then, is in the form of a Conceptual Dependency structure. The meaning of a word in the lexicon of CGEN is a Conceptual Dependency frame, with restrictions on the fillers of various slots. The ordering of concepts to be expressed is obtained from syntatic predicates, which are indexed by the filler of the FOCUS slot for full conceptualizations. In addition to ordering information, the predicates specify function words. The following sections describe, in detail, how CGEN uses this information to produce grammatical English sentences, and describes several modifications made to it for

the current work.

2.3 The Basic Lexicalization Process

When a concept is given to CGEN represented in CD format, a basic cycle performs the lexicalization and sequencing. This part of the generator transforms the input conceptualizations into a string of English words by repeatedly looking up words and organizing the words found, the slot fillers not spanned, and any function words needed according to the syntax stored with the words.

CGEN's control structure is similar to the Conceptual Analyzer of Birnbaum and Selfridge [Birn81]. It has a short term memory, called the C-LIST (Concept List) which stores concepts that need to be generated, and words that need to be said. The focus is always on the top of the C-LIST. If the top of the C-LIST is a word, CGEN says it. If it is a concept, it is examined by a set of rules (discussed in the next section) which may modify the concept and the rest of the C-LIST. The item on top of the C-LIST is then sent off to a dictionary specialist.

Words in the dictionary are organized according to the conceptual class and slots of the CD frame that is their basic meaning. The entire set of words CGEN knows is stored in a discrimination tree [Jose83]. The tree orders the dictionary entries from most to least specific. A dictionary entry matches if it is structurally similar to the concept to be expressed. Structurally similar means that the two have the same slots, and that any non-nil slot filler in the diction-

ary entry matches the same filler in the concept to be matched. Once the structural criteria have been fulfilled, the semantic restrictions on the slot fillers are checked. If all these restrictions are met, a word has been found.

The word found may or may not span all of the concept to be expressed (e.g. bachelor spans the male meaning of the unmarried man concept, making male bachelor redundant). Therefore, the next step in the dictionary lookup is the packaging of those slot fillers that still need to be expressed along with their syntatic predicates and function words. The notion of precedes and follows can be used to describe any ordering used by CGEN. In particular, it can be used to specify syntax where the focus is on any filler in the concept. Multiple syntatic rules are stored under each word that constitutes a main action. When a concept is sent to the generator, it will contain a property specifying what subpart of the whole concept is to be focused upon. This focus property is used to index the syntatic rules specifying the ordering of concepts (as in Figure 2.2). Note that this focus is not part of the concept proper, it is more of an indication of the intent of the speaker. It was found to be necessary for the CADHELP domain (and was thus added as an extension to CGEN) since explanations which involved the CAD system as actor frequently would not express the actor (e.g. "The device is deleted from the design").

Once the proper set of syntatic predicates has been found, each non-nil slot filler in the input concept that matched a nil (don't-care) slot filler in the dictionary entry is given a set of syntax

that specifies its position on the C-LIST relative to the word found and the other fillers. Since the function words associated with a slot appear in the set of of syntatic predicates that are associated with a filler if the filler is not present, they will not be said. The function words, when encountered, obtain their syntax from the slot they are associated with, in addition to having a predicate specifying where they are to be placed relative to the filler of that slot. For example, the function word from generated using the dictionary definition from Figure 2.2, would have the following syntax:

((fo parent) (fo (path MOBJ)) (pr (path FROM)))

Some slot fillers receive special treatment at the hands of the syntax specifier. These fillers fill surface slots, slots in a concept that do not contain a substantive concept as a filler, but a lexical item that can be expressed without further dictionary lookup. Examples of these are the names of people, names of commands in CAD systems, numbers and titles of courses, etc. The surface slots for a given concept can be determined from the conceptual class (e.g. for the conceptual class PERSON, the surface slots are PERSNAME and SURNAME). Fillers from slots such as these that need to be expressed are returned as words to insert onto the C-LIST, not as concepts that need to be generated.

The dictionary is responsible for selecting the word or words that span as much of the concept as possible. If the concept sent to the dictionary contains temporal or modal information, a verb cluster must be built to express the TIME and MODE slots of the concept. This information is like the focus information, i.e. it is not an integral

part of the meaning of the concept since it expresses auxiliary information. In fact, the three types of information, focus, time and mode, have to do with issues that are outside the range of the single concept being generated. The time information expresses the relationship of the time the action or state in the concept occurred, relative to the time of speech and possibly relative to the time of some other event. The focus and modal information relate the intentions of the speaker or hearer. Focus serves to call attention to a particular part of the concept, and modal information expresses the ability, intent, obligation, etc. of the speaker. This information is complex and difficult to represent fully. For example, the modal concept expressed by should in: "You should take CS 110 next semester." really refers to the fact that the speaker has some knowledge of what events would be in the best interest of the listener. This complex notion is condensed into a single word, should.

Tensing, aspect and modal expression as well as subject-verb agreement are performed in the lookup routines whenever a main verb is found to match the concept sent. Subject-verb agreement is done by examining features of the focused on concept to determine if it is first person, second person, plural, etc. This information is used by CGEN to produce the grammatical form for sentences like:

8) Move the stylus to the tablet.
9) The cursor moves.
10) I teach CS 110.
11) Dr. Bernard Lovell teaches CS 267.

The generation of modals is done by translating a given modal specifier into the corresponding modal word (e.g. urge becomes should, ability becomes can). In addition, those modals that can indicate tense (can and shall) do so. If the concept is negated, this is also specified in the mode slot, and CGEN adds the word not to the verb cluster immediately preceding the main verb, and following any modal word. If the word not is used, and no modal is present that can carry the tense, then the properly tensed form of do is added (called do-support). For example, if the verb found is go, the tense is past, and the mode slot filler is:

MODE (MODES MODE1 (negation) MODE2 (ability) MODE3 (nil))

the dictionary lookup routine will return could not go. However, if the mode slot filler is:

MODE (MODES MODE1 (negation) MODE2 (nil) MODE3 (nil))

and the lookup routine returns did not go.

The third slot in the mode slot filler of a full conceptualization, MODE3, is used if the entire conceptualization is being questioned. If the MODE3 slot does contain the question marker, then depending upon the tense, aspect, MODE1 and MODE2 fillers, several things are done. First, the verb cluster is formed as usual, with the exception that, if there is no modal specifier in MODE2, and no aspect, do-support is done to carry the tense, since the first auxiliary (is, had, etc., or a modal) will be fronted to form the question. Recall the definition of delete, given in Section 2.2. This definition allows CGEN to generate sentences like 12 and 13.

12) The system deletes a device.
13) The device is deleted by the system.

If CGEN is sent the question form of the above concepts, the result will be 14 or 15.

14) Does the system delete a device?
15) Is the device deleted by the system?

In sentence 15, the verb cluster _is deleted_ is split and the auxiliary _is_ is placed before the focused on entity. In order to do this, in the case of a question, the first auxiliary is returned as a word to be inserted into the C-LIST, not as a part of the verb cluster. The syntax for the auxiliary is formed by specifying that it precedes the focused on concept. In general, the auxiliary that is moved may be a modal, a form of _do_ generated by do-support, or the copula of a progressive or stative (e.g. "Is the chair on the table?").

If only a subpart of the full conceptualization is being questioned, for example, the identity of the actor is desired, and the concept is generated in such a way that the questioned part is generated before anything else (see Section 2.4), the auxiliary must also be moved. This produces:

16) What course are you taking?

instead of

17) What course you are taking?

In sentence 17, by the time the dictionary is told to look up the case frame that will match the word _take_, the questioned part of the con-

cept, what course has already been said, so are is returned from the lookup routine with syntax specifying that it is to be placed before the filler of the ACTOR slot. If, at lookup time, the questioned concept has not been expressed, no fronting of the auxiliary will be done, and CGEN will generate 18.

18) You are taking what course?

The interaction between tense, aspect, modals, negations, questions and focus is intricate. In the simplest (and most common) case, only tense and aspect are specified and these are handled easily enough. CGEN has a set of morphology routines capable of adding ed, ing, s, to words, and stores any irregular past tense or participle forms under the root word. Some examples are shown below.

18) I took CS 100.
19) I am taking CS 110.
20) I had taken CS 110.
21) He has been teaching CS 110.
22) Should he have not been teaching CS 110?

The net result of the dictionary lookup and its associated processing is a word or list of words that represent as much of the input concept as possible. In addition, it returns a (potentially empty) list of concepts that still need to be expressed, and function words. After these concepts and words are inserted into the C-LIST using the specified precedes and follows predicates, CGEN repeats the cycle by examining the top of the C-LIST.

Reviewing some of the examples given in this section, there appear to be some phenomena CGEN can produce that are still

unexplained. For example, what happened to the expression of the actor in sentence 8 ? How is the filler that is being questioned in a full conceptualization fronted in sentence 16 ? The processes that supplement the basic cycle to produce these sentences and others are discussed in the next section.

2.4 Generation Strategies

The process described above is not powerful enough to always produce natural-sounding English. There are a number of conventions speakers use to produce utterances that are as concise as possible, without omitting important information. These conventions have been added to CGEN in an *incremental manner*, as needed. For a CAD style domain, where the user is being told how to perform physical actions, the need for the imperative and expressions of instrumentality become apparent. Upon changing to the domain of interviewing a student to perform a preregistration, an important task for the generator was to produce questions. The conventions for producing these are implemented as a set of rules (sketchifiers) that fire when certain semantic features are present in the concept on top of the C-LIST. These rules may change the form of the utterance by marking redundant concepts as not to be said, by adding words to the C-LIST, or by modifying the concept so that certain linguistic conventions are used (such as the progressive) to express a concept more concisely than the normal cycle would. It is this set of rules that is responsible for forming the linguistic shorthand used by speakers of English.

The sketchifiers CGEN uses are arranged according to the type of concept upon which they operate. One set operates on entities. It contains a rule which expresses relative clauses to describe an entity, if need be. If CGEN has received an entity for expression that contains a relative clause, and if the entity has not been mentioned before, the SAYREL sketchifier will remove the relative concept from the concept representing the entity and place it on the C-LIST following the entity representation. This could be done in the dictionary just as easily, using some syntatic predicates to order the expression of the entity relative to the expression of the relative clause. However, SAYREL remembers that it has expressed the entity using a relative clause, and will express the relative clause slightly differently the next time it appears. If the time is present in the relative clause, and the mode is true, then the connecting pronoun that and the copula that connects the pronoun to the rest of the clause can be removed. For example, the phrase:

"the device that is in the lower right hand corner of the screen"

can be shortened to:

"the device in the lower right hand corner of the screen."

Note that if the time is not present or the mode is not true this does not hold, as in:

"the device that was in the corner"

or:

"the device that should be in the corner."

In addition, if SAYREL sees a concept that has a LABEL attribute as its relative clause, e.g. "the command block labeled MARK1," it knows

this can be shortened to a simple expression of the label, MARK1.

The other rules that operate on entities are used to fill default values in the case frame for an entity. Entities can be referenced in a variety of ways, with definite or indefinite articles, or using pronouns, as well as using the relative clauses described above. In a system with a sophisticated memory model, any reference specifications would be filled by the memory before being sent to the generator or the generator would keep track of the different ways an entity can be expressed. One system that has been implemented for use with CGEN is the paraphrase module for the DSAM story understanding program [Cull81a], [Unge82]. The DSAM system tracks the expression of people and when a new person appears, creates a new sketchifier whose specific task is to determine how that person should be expressed each time it occurs in a concept. For entities other than persons, the sketchifier ENTREF was built into CGEN by Cullingford, et. al., and is responsible for filling in the slot that determines how an entity is to be referenced. That part of the case frame representing an entity that determines the reference for an entity is the REF slot. ENTREF is responsible for choosing between three simple types of expression. First, CGEN retains a list of those entities that have been expressed in the current sentence. If an entity appears more than once, the second time it is expressed as it. This simple rule allows CGEN to produce simple pronominal reference for entities. If the entity has not been said in the sentence, but has been mentioned in the text or is a unique entity (one that the listener could be expected to know from the situational context) it is given a definite reference. The

default reference is indefinite. Thus, ENTREF will yield the following phrases in a run:

```
a device will appear
move the device
select the device and move it
```

The next set of rules operates on that class of concepts that express some relation between full conceptualizations, for example, causal or time relations. One causal relation that occurred frequently in the CAD domain was one in which the first concept expressed some goal of the user, and the second concept stated the action the user had to perform to realize that goal. For example, if the concept was the intent of the DELETE command, it would look as shown in Figure 2.3. Given this concept, CGEN's basic cycle will produce: "If you want that you delete a device from the design, then you use the DELETE

```
(CAUSE PRECON (S-GOAL ACTOR *user
                      MODE (nil) TIME (nil)
                      GOAL (MTRANS
                             ACTOR (PERSON PERSNAME (cadhelp)
                                           ROLE (*sys))
                                   MOBJ &desr-del-dev
                                   INST (nil) FOCUS (nil)
                                   MODE (nil) FROM *design
                                   TO (nil)))
       POSTCON ($CADFEAT ACTOR *user
                         FEATNAME (CREATE)
                         MODE (nil) TIME (nil)))
```

A Causal Relation Input to CAUZ

Figure 2.3

command." However, the CAUZ sketchifier will notice this particular causal relationship and produce: "To delete a device from the design, use the DELETE command." CAUZ does this by adding the word to to the C-LIST, followed by the concept filling the GOAL slot of the S-GOAL of the user and the POSTCON of the CAUSE. This process is shown in more detail in the example in Section 2.5.

The last set of rules to be discussed has to do with modifications to concepts that are full conceptualizations. One such sketchifier is IMP, the imperative rule. IMP demonstrates the complexity of the decisions some of these rules must make before they can fire. An English version of IMP appears in Figure 2.4. This rule produces an imperative form whenever all the conditions are true. Another sketchifier that operates on full conceptualizations, INF, is responsible for forming an infinitive construction whenever an actor has a goal for himself, or in certain cases where mental events are

---

```
If the concept is a unit action
(i.e. not a state or relation between concepts)
and If the actor of the action is the other person
    in the conversation
and If the time of the action is present
and If the action is asserted, with no negation
    or modal specifiers
and If the concept is not a question
and If the focus of the expression is to be on the actor
     then suppress the expression of the actor.
```

The Imperative Rule

Figure 2.4

---

occurring. For goals, INF will take a concept that would normally be expressed as: "Jacob wanted that Jacob go to Hartford" and express it as: "Jacob wanted to go to Hartford." For mental events such as are found in the meaning of a sentence like: "Jacob pretended that Jacob went," the INF sketchifier will produce "Jacob pretended to go," but will skip over a concept that would express "Jacob pretended that Jacob could go," since the infinitive form does not occur here.

An important feature of the CAD domain was the use of instruments. Concepts would frequently have the actor producing some action by performing some instrumental action. In concepts such as these, the instrument can be expressing by suppressing the expression of the actor in the instrument action (since it can be inferred), then using the word by followed by the progressive form of the instrument action. This causes those concepts which would normally be expressed like: "Jacob hit the ball by the instrument that Jacob swung the bat." to be instead shortened to: "Jacob hit the ball by swinging the bat." Further uses of this instrument will be shortened to: "Jacob hit the ball with the bat." provided that the entity serving as the object of the instrumental conceptualization is being used in its usual manner.

The last rule to be discussed is responsible for forming questions, and is called the QFOCUS sketchifier. If the entire concept is being questioned, the dictionary deals with the ordering of words, etc, as described above. Otherwise, the proper form of a wh-word (for CGEN's applications, these are who or what) is determined. This is done by examining the concept being questioned to see if it is a per-

son, and that it has a place in the being generated. Then the form used is who. If who is not enough to completely specify the reference to the entity being questioned, the word what is used followed by any information that can be obtained from the concept. For example, if CGEN was generating a question directed to a user of a program, a proper form would be: "Who are you?" On the other hand, non-person entities must be stated in full, e.g. "What courses are you taking?"

CGEN was developed and expanded initially for generating explanations in a CAD domain. In addition, it performed in the story paraphrase task of DSAM [Unge82]. It was also successfully updated for use in the Academic Counseling Experiment. Once the basic cycle was fully developed, CGEN could move from domain to domain by adding new words to the dictionary, and by adding new sketchifiers to produce new forms. This important tool is used in both systems discussed in this thesis, and thus has had an influence upon their design. Before further discussions of these systems, a full-length example of CGEN's operation is shown in the next section.

2.5 Detailed Example

CGEN's generation cycle and rules are here illustrated with an example. What follows is annotated computer output, edited for readability, showing the generator expressing a concept which has been modified by several sketchifiers. The concept to be expressed is an intent conceptualization of one of CADHELPs commands, the CREATE command. In a fully verbose form, with no rules to supplement the basic cycle, the generator would say:

"If you want that CADHELP add a device to the design then you use the CREATE command."

With the rules present, CGEN produces instead:

"To add a device to the design use the CREATE command."

A trace of the generation process is shown below. Comments added in to clarify the example are set off by horizontal lines, or by tildes "~".

```
Franz Lisp, Opus 36
-->(gen 'create-intent)
```

If there's a concept at the top of the clist, CGEN will print it before invoking the sketchifiers. The current top of clist is the input conceptualization, the intent of the CREATE command. It expresses a causal relation between two events, a precon and a postcon. The precon concept is that the user has the goal that the system place a device (which is in the warehouse) in the design. The system accomplishes this transfer by means of an mtrans, a mental transfer of information from the warehouse to the design. The postcon concept is a script, $cadfeat, used to represent the complex notion of "executing a CADHELP feature". In this case, the CADHELP feature to be executed is CREATE.

```
GEN: top of clist
     (cause
        precon (s-goal actor *user
                       mode (nil) time (nil)
                       goal (mtrans actor *sys
                                    mode (nil) time (nil)
                                    mobj *disp-dev
                                    inst (nil)
                                    from *wrhouse
                                    to *design
                                    mode (nil)
                                    time (times time1 (:pres)))
        postcon ($cadfeat actor *user
                          featname (CREATE)
                          mode (nil)
                          time (times time1 (:pres)))))
```

Now this concept is sent to the sketchifiers. The first one to fire is the causal sketchifier. This forms the construction "To x, y" from the concept "If you want that the system x, y." It does this replacing the precon of the input concept with its goal subconcept, preceded by the infinitive function word "to." It also marks the actor of this subconcept as not needing expression, since in this domain the system can be inferred to be the actor in concepts of this sort.

```
*cauz
 cauz: forming to construction in top of clist
```

The state of the clist at this point is:

```
GEN: clist
  ("to"                                  ~ word "to"
  (mtrans actor *sys                     ~ concept that was goal of
          mode (nil) time (nil)          ~ s-goal of the precon
          mobj *disp-dev
          inst (nil)
          from *wrhouse
          to *design
          mode (nil)
          time (times time1 (:pres)))
  ($cadfeat actor *user                  ~ postcon concept
            featname (CREATE)
            mode (nil) time (times time1 (:pres))))
```

CGEN pops the word "to" off of the top of the clist and saves it. Since the next thing on the clist is a concept, CGEN will print it, then let the sketchifiers look at it.

```
GEN: top of clist
     (mtrans actor *sys
             mode (nil) time (nil)
             mobj *disp-dev
             inst (nil)
             from *wrhouse
             to *design
             mode (nil) time (times time1 (:pres)))
```

The concept is sent to the dictionary for lookup. The dictionary returns the lexical item "add", since the direction of the transfer is from the warehouse to the design. Notice that the actor of the mtrans, the system, is not among the fillers returned, since it was marked as not to be said by the CAUZ sketchifier.

```
DICT to match:
     (mtrans actor *sys
             focus (actor)
             mode (nil) time (nil)
             mobj *disp-dev
             inst (nil)
             from *wrhouse
             to *design
             mode (nil)
             time (times time1 (:pres)))


DICT result: (add)
DICT fillers: (*disp-dev              ~ mobj slot filler
               (follows "add")
               (precedes to)

               *design                ~ to slot filler
               (follows "add")
               (follows mobj)
               (follows (fw to)))     ~ fw = function word
```

The clist after insertion is:

```
GEN: clist
     ("add"                            ~ new word, "add"
      *disp-dev                        ~ mobj filler of the mtrans
      "to"                             ~ function word "to"
      *design                          ~ to filler of the mtrans
      ($cadfeat actor *user            ~ postcon of the causal
                featname (CREATE)
                mode (nil) time (times time1 (:pres))))
```

The new top of the clist is the device, *disp-dev. It is shown below in its expanded form; *disp-dev is a shorthand form.

```
GEN: top of clist
     (#device partof *wrhouse
              type &typ
              class &cls
              posx (1471)
              posy (144)
              status (nil)
              assoc-txt (nil)
              label &label)
```

The job of the next sketchifier is to track the entities that have been said, and see to it that they are given the appropriate reference. In this case, since the device has not been mentioned before, it is given an indefinite reference.

```
*entref
 entref: refizing top of clist


  DICT to match:
       (#device partof *wrhouse
                ref (indef)          ~ indefinite marker
                type &typ
                class &cls
                posx (1471)
                posy (144)
                status (nil)
                assoc-txt (nil)
                label &label)
    DICT result: (device)
    DICT fillers: ((indef)           ~ ref slot filler
                   (precedes "device"))
```

The clist after insertion is:

```
GEN: clist
     ((indef)                          ~ ref slot filler
      "device"                         ~ new word
      "to"                             ~ function word "to"
      *design                          ~ to filler of the mtrans
      ($cadfeat actor *user            ~ postcon of the causal
                featname (CREATE)
                mode (nil) time (times time1 (:pres))))
```

The concept indef is found to match the word "a".

```
GEN: top of clist
     (indef)


DICT to match:
     (indef)
DICT result: (a)
DICT fillers: (nil)                ~ no fillers
```

The clist after insertion is:

```
GEN: clist
     ("a"                          ~ new word
      "device"
      "to"
      *design                      ~ to filler of the mtrans
      ($cadfeat actor *user        ~ postcon of the causal
                featname (CREATE)
                mode (nil) time (times time1 (:pres))))
```

The next concept to reach the top is *design, the shorthand way of naming the design. Entref gives it a definite reference, since it is a known entity (like "the screen" and "the user"). In the same way as for "a device", "the design" is generated. We shall skip the details and go to the generation of the postcon of the causal.

```
GEN: top of clist
     ($cadfeat actor *user
               featname (CREATE)
               mode (nil) time (times time1 (:pres))))
```

Here the imperative sketchifier, IMP, goes off, since the concept at the top of the clist is a simple declarative. The actor in the concept is marked as not to be expressed.

```
*imp
 imp: squashing actor in top of clist

    DICT to match:
         ($cadfeat actor *user
                   featname (CREATE)
                   mode (nil) time (times time1 (:pres))))
    DICT result: (use)
```

The dictionary has not been able to find a verb which directly expresses the content of the concept above, so it returns the neutral form "use." "Execute" or "do" are other possibilities. The only filler returned is a nominalized form of the $cadfeat script. This is used to represent the generic term "command," which in this domain is a complex series of events.

```
DICT fillers: (($cadfeat actor *user
                        featname (CREATE)
                        mode (nom) time (nil))
               (follows  "use"))
```

The clist after insertion is:

```
GEN: clist
     ("use"                          ~ new word
      ($cadfeat actor *user          ~ nominalized script
                featname (CREATE)
                mode (nom)))
```

The nominal form is at the top, and is sent to the sketchifiers. A rule similar to ENTREF, the entity reference rule, fires. This is EVREF, the event reference rule. EVREF gives the feature a definite reference, since the user supposedly knows about the commands.

```
GEN: top of clist
     ($cadfeat actor *user          ~ nominalized script
               featname (CREATE)
               mode (nom) time (nil))

*evref
 evref: refizing top of clist
```

```
DICT to match:
     ($cadfeat actor *user          ~ nominalized script
               ref (def)
               featname (CREATE)
               mode (nom))
DICT result: (command)
DICT fillers: ((CREATE)            ~ featname filler
               (precedes "command")
               (follows ref)

               (def)               ~ ref slot filler
               (precedes "command")
               (follows featname)
```

Notice in the clist that follows, the filler of the featname slot has been put on as a word, not as a concept. This is because certain concepts have certain roles that are 'labels' (such as the names of persons and animals), and do not need further lookup.

```
GEN: clist
     ((def)                        ~ ref slot filler
      "CREATE"                     ~ featname slot filler
      "command")                   ~ new word

GEN: top of clist
     (def)

DICT to match:
     (def)
DICT result: (the)
DICT fillers: (nil)

GEN: clist
     ("the"
      "CREATE"
      "command")

GEN result: (to add a device to the design use the CREATE command)
```

CHAPTER 3

Explanations in Computer-Aided Design

3.1 Overview of CADHELP

CADHELP is a computer-aided design system for the design of logic circuits [Cull81c]. As discussed previously, a CAD domain is a useful vehicle for experiments in explanations. The CADHELP system is divided into two basic parts, the CAD Tool itself, which performs graphical operations concerned with the design and the Explanation Mechanism, which is responsible for providing natural language and graphical output to the user. The function of the Explanation Mechanism is to explain how a particular feature works, and to assist in the execution of a feature by generating prompts to guide the user. In order to facilitate understanding of the examples included in this chapter, it will be useful to discuss the actual features the CAD Tool can execute.

CADHELP operates in the task domain of logic circuit design. The graphics component of CADHELP provides the user with the ability to select, place, and orient components on a graphics screen, and to make connections between devices. A user can also edit a design by adding, deleting, or moving components and adding, deleting or redrawing interconnections. The system provides a technique for creating connections with right-angle segments, as well as a mechanism for commenting on the design by associating text with a particular device or

a special comment symbol.

The main channel between the user and CADHELP is a 20"x20" data tablet and its associated pen-like stylus. The surface of the tablet is divided into a Drawing Area, a Master Control Block, and 64 permanently allocated 1"-square Command Blocks. Touching the tip of the stylus to the tablet communicates coordinate information to the system. Additionally, the tip of the stylus contains a switch, which is turned on if the stylus is pressed sufficiently hard. Pressing the stylus on the Master Control Block will abort any ongoing command. Exiting a command via the Master Control Block, as well as other normal terminations, returns the user to the top level of the system, where another command may be selected for execution. The Drawing Area is used for a variety of input functions, such as drawing interconnections and moving graphical objects on the screen. The Command Blocks on the tablet are used to select and control the execution of the graphical features of the CAD system.

The commands currently implemented and known to the Explanation Mechanism are outlined in Figure 3.1. It is not at all obvious how these features are to be operated, especially to new users. The CREATE command will be described in some detail below as an illustration of the complexity of the commands.

CREATE is used to select a device from CADHELP's database of devices, called the warehouse, and position it on the screen. This is how new devices are added to the design. First, the user peruses the warehouse, looking for the device to be created. The perusal process

SELECT
This is CADHELP's top level. Any command can be initiated by touching the stylus to the Command Block labeled with its name.

CATALOG
allows the user to peruse CADHELP's database of logic devices.

CREATE
select a device and position it in the design area.

CONNECT
draw a connection between devices containing right-angle segments. This feature uses a simple extension of the graphical operation called rubber-banding, in which a line segment appears to stretch away from an origin in response to stylus movements.

DELETE
delete a device from the design.

DISCONNECT
delete an interconnection.

DRAG
move an existing symbol.

ROTATE
orient a device symbol left, right, up or down

ANNOTATE
associate text with particular device or comment symbol.

READ
read text associated with a logic component or comment symbol.

CADHELP's Command Summary

Figure 3.1

is implemented by a feature called CATALOG. To catalog, the user touches the stylus on the Drawing Area of the tablet. The CAD Tool responds by drawing a device in a dedicated area of the display, the catalog area. If the user now moves the stylus horizontally in the

Drawing Area, a new device will appear which is of the same class as the device currently being displayed, but of a different type (e.g., 2-input vs. 3-input NAND gates). To view a member of a different class (e.g., a counter vs. an OR gate), a vertical movement of the stylus is made.

When the device the user wants to create finally appears in the catalog area, he informs the CAD Tool of his choice by pressing the stylus on the command block labeled MARK1. Pressing with some force is necessary to activate the switch in the stylus which means: "attend to this command." The Tool then makes the device being displayed movable. Now the user must position the device in the design. To enable the user to locate the device with the stylus, the system draws a cursor which moves on the screen as the user moves the stylus on the drawing area. The user moves the stylus, and thus the cursor, in the direction of the device to be added. When the device and the cursor overlap, the device also begins to move as the stylus moves. By moving the stylus appropriately, the user positions the device. When the device has reached the desired spot, the user informs the system of his decision by pressing the MARK1 command block.

CADHELP uses the Explanation Mechanism to describe features such as CREATE both when the user SELECTs the command EXPLAIN and through prompts during normal operation of a feature. After choosing to execute EXPLAIN, the user will be asked to press the command block that is labeled with the name of the feature to be described. He will also be asked to select the level of explanation desired by touching one of

the blocks labeled SUMMARY, NORMAL or ERRORS.

In a summary level of explanation, the intent of the command is given. The intent of a command is the result that execution of that command will produce. For example, the intent or goal for CREATE is:

To add a device to the design, use the CREATE command.

The normal mode of explanation provides the user with a step-by-step description of his expected behavior and the response of the CAD Tool to that behavior. In addition, system features and components that have not been mentioned in other explanations are described (e.g. the catalog area in the output shown below). For example, the first time the user requests a NORMAL explanation of the CREATE command, CADHELP responds with:

> To add a device to the design,
> use the CREATE command.
>
> Move the stylus to the tablet.
> Touch the stylus on the drawing area.
> A device that is in the catalog area will become visible.
> The catalog area is in the lower right hand corner of
> the screen.
>
> Repeat the following until the device in the catalog area
> is of equal type to the device that you want to add to the
> design.
>
> Move the stylus horizontally.
> A new type of device will become visible.
>
> Repeat the following until the device in the catalog area
> is of equal class to the device you want to add to the
> design.
> Move the stylus vertically.
> A new class of device will become visible.
>
> Move the stylus to the command block that is labeled MARK1.
> Press the stylus on the command block labeled MARK1.

A prompt will become visible.
A cursor will become visible.
The cursor is in the lower right hand corner of the screen.

Move the stylus to the tablet.
Touch the stylus on the lower right hand corner of the drawing area.

Repeat the following until the cursor is over the device in the catalog area.
    Move the stylus to a new location.
    The cursor will move to a new location.
    The new location will correspond to the location of the stylus in the drawing area.

Repeat the following until the device is at a screen location that you want that the system record.
    Move the stylus to a new location.
    The device will move to a new location.

Move the stylus to the MARK1 command block.
Press on MARK1.
The device will be added to the design.

An ERRORS explanation is like a NORMAL one, except that the Explanation Mechanism also describes what can go wrong during the execution of the feature. For details on the ERRORs mode of explanation, see [Phel82]. For example, while the user is moving the cursor toward the device in CREATE, he may move the stylus outside of the Drawing Area. The Explanation Mechanism will explain this potential error as follows:

.
.
.
Repeat the following until the cursor is over the device in the catalog area.
    Move the stylus to a new location.
    The cursor will move to a new location.
    The new location will correspond to the location of the stylus in the drawing area.
    If you move the stylus out of the drawing area

the location of the stylus on the tablet will not correspond to a location on the screen.
The cursor will not move.
.
.
.

The actual use of the graphical features during design is accompanied in CADHELP by prompts which are intended to lead the user step-by-step through the operation. The prompts are very much like the NORMAL mode of explanation illustrated above. Unlike the prompts provided with existing CAD systems, however, these are not canned. They are generated from the knowledge structure each time they are expressed. Thus, the system is verbose with a new user but becomes more and more laconic as it gets out of the way of the experienced designer.

For example, the first time the user operates the CREATE command, the Explanation Mechanism provides the CAD Tool with a sequence of prompts which is nearly identical to the NORMAL explanation shown above, minus the first sentence which expresses the intent concept. The only difference is that the EXPLAIN command produces the future tense in expressing the actions of the CAD Tool, whereas the prompting mechanism uses the present tense, since the system's actions are occurring in real time. If the user operates CREATE a second time, CADHELP generates a more abbreviated prompt sequence:

Move the stylus to the tablet.

To see the warehouse move the stylus horizontally and move the stylus vertically.

> To tell the system to add the device to the design press the stylus on MARK1.
>
> Move the stylus to the lower right hand corner of the drawing area.
>
> To move the cursor move the stylus.
> To move the device move the stylus.
>
> To tell the system to record the screen location press on MARK1.

The third time CREATE is used, CADHELP generates the following simple prompt sequence:

> Move the stylus horizontally and move the stylus vertically.
> Press on MARK1.
> Move the cursor with the stylus.
> Move the device with the stylus.
> Press on MARK1.

Thus CADHELP's explanations become more brief as the user gains experience.

In order to generate language and graphical animation for CADHELP features as complicated as CREATE, a complex representation of the command as an expert sees it is needed. This representation and the mechanism that operates upon it, are discussed in the following sections.

3.2 Representation of CAD Knowledge

In order to explain a command as complex as CREATE in CADHELP, the Explanation Mechanism must have knowledge of the execution of the command as an expert user of the system sees it. This viewpoint (as opposed to the expert knowledge the developer of the system has) will

yield the best explanations. The knowledge an expert user has about a system such as CADHELP is best represented in terms of the give and take between the user and the system. Several theories exist for structuring knowledge. Artificial Intelligence programs with large databases containing both declarative and procedural knowledge have found the production system approach useful [Shor76]. Production systems consist of sets of rules represented as test-action pairs. The test part of a rule stands for conditions (the declarative knowledge) which, if satisfied, will perform some action (the procedural knowledge). Since the knowledge CADHELP's Explanation Mechanism used was to be more declarative than procedural, a representation that reflected the static aspects of knowledge was desired.

For the reasons outlined in Chapter 2, the set of concepts comprising the CAD expertise were to be represented using Conceptual Dependency format. However, these concepts needed to be causally connected to exhibit the stereotypical behavior of a user of the system. One theory for structuring complex, stereotyped knowledge is the frame-system proposed by Minsky [Mins75]. Charniak [Char77] used the notion of frames in implementing a language comprehension system called Ms. Malaprop. Ms. Malaprop specialized in understanding stories about mundane painting tasks. Charniak uses the frame representation to list goals that can be achieved by realizing subgoals, but is not explicit about the actions involved in realizing the subgoals or the causal connection between events in a frame, both of which are necessary for the level of detail to be represented here.

A similar knowledge structuring technique was used by Cullingford [Cull81a] in his SAM story comprehension system. Cullingford's SAM understands stories about stereotyped events using a knowledge representation called scripts [Scha77]. A script is a causally connected set of concepts (in CD format), which models the knowledge people have of stereotyped situations, such as eating in a restaurant or riding on the subway. While scripts have the advantage here since they are based on CD format, the scripts that SAM used for understanding were not fine-grained enough for representing the use of the commands of CADHELP.

Detailed knowledge is represented best in the commonsense algorithms of Rieger [Rieg77]. The primitive concepts of Conceptual Dependency, linked into a script by the links of the Common Sense Algorithms, provide a static representation that is fine-grained enough to permit a graphical animation system to use them, and not so fine-grained that a language system is caught up in unnecessary details.

Each command in CADHELP is represented as a separate feature script stored in LTM (Long Term Memory). A feature script is composed of the CD representations of the physical and mental actions and states of the user and the system, causally linked together. The links used are listed in Figure 3.2. Detailed discussion of the links is deferred until the next section. Figure 3.3 shows the primitive actions and states used in the feature scripts. In addition to those primitive actions listed, some primitive embedded scripts were used

OSE:
a state one shot enables an event; it must be present once for the event to occur

OSC:
an event one shot causes some states; it need not continue to be performed in order for the states to still exist

CC:
one state is causally coupled to another; the two are causally connected but the exact nature of the causality is not specified

INITIATE:
an event (usually a perception of some state in the world) initiates another event (ususally a mental event)

REASON:
a mental event is the reason for another (ususally physical) event

IR:
this blurs the distinction between an REASON and an INITIATE link, when this information is not useful

RUT:
repeat links until threshold (satisfaction condition) becomes true

TRNPT:
indicates a turning point in the script, a set of mutually exclusive paths which can be followed

SR:
performing the acts comprising a script leads to some important states

GRC:
the overall goal of the script (a state) is linked by this to the action that caused it

ANTAG:
an antagonism between two states exists

Causal Links Used in Feature Scripts

Figure 3.2

PTRANS:
the physical transfer of location of an actor or entity by an actor

MTRANS:
the communication of concepts between actors or the acquisition of knowledge from a sensor

MBUILD:
construction of a decision out of pre-existing information, retrieved from memory or a sensor

PROPEL:
application of force to an object with another object

MKNOW:
the state of having some information in memory

S-CHANGE:
a change of some state of an actor or entity

S-EQUIV:
a equivalence between entities

P-CONFIG:
specifies the physical configuration between two entities

A-CONFIG:
specifies an abstract configuration between two entities

Primitive Actions and States Used in Feature Scripts

Figure 3.3

(see Figure 3.4). These occurred in two cases. First, if several feature scripts shared common actions and states (perhaps differing only in the entities manipulated), these common concepts could be represented once, and used by all feature scripts. This makes the feature scripts more concise, and makes for easier development of new scripts. For example, one common sequence is that of pressing the

stylus on a command block on the tablet, and this is represented as the $PRESS embedded script. Hence, any feature script needing to express this feature could refer to $PRESS. Since there are several command blocks that could be pressed with the stylus, the $PRESS script uses a script variable, &cmdblk, to represent the command block to be pressed, and any feature script using it will instantiates &cmdblk with the actual occurrence of a command block, for example *MARK1. The notation *MARK1 is a convenient shorthand for the complicated concept representing the MARK1 command block on the data tablet. It is expanded when encountered in any concept to its full form.

The embedded script representation is usually used for those actions performed by the user. There is a similar embedded script representation for those actions performed by the system. These are represented in the same way as the user scripts, but are not expandable into a causal chain of actions and states. This shortening is done for several reasons. Firstly, the view of the CAD features modeled in CADHELP is the expert user's view, and the expert user is aware that the system performs complicated actions in the form of code, but is not aware of the exact details. The expert user is only aware of the consequences of those actions, and this is represented explicitly. Secondly, the graphical animation expert using the representation did not need to know the details either, if this expert knew the desired result, it used its own code to depict that result on a graphics screen. A third point concerns whether or not to simply make these scripts primitive actions in the system. This was not done because of a wish to emphasize the complex nature of the action, and

Non-expandable embedded scripts:

$prompt
the system executes this to cause a prompt to appear on the screen

$clone
used to make a copy of a device or other graphic object

$draw
the system makes a line or device visible on the graphics display with this script

$undraw
the system makes a line or device invisible on the graphics display with this script

$makemap
the system forms a correspondence between two objects

$cadfeat
this script is used to refer to any other CADHELP feature, without necessarily specifying which.

Expandable embedded scripts:

$press
used to inform the system of a user intention by the user pressing the stylus on a command block

$move
used to move a graphical object on the screen by moving the stylus along the corresponding points in the drawing area.

$viewrhs
used to view the contents of the warehouse by moving the stylus in the drawing area

Embedded Scripts Used in Feature Scripts

Figure 3.4

also to leave open the possibility that in an extension of the system, these scripts may become expandable.

CADHELP's LTM, then, contains a knowledge base of feature scripts, one for each command CADHELP knows. In addition, CGEN can express the feature scripts in English. The resulting output would be difficult to understand, because of all the unnecessary detail present. The next section outlines the interface program between LTM and CGEN which alleviates this problem.

3.3 Concept Selection

The rules that occur in CGEN are decision rules, they decide, based on the semantic features of the concept being expressed, what linguistic form to use to communicate the idea in the most economical fashion. There are also other rules that are distinguished from CGEN's in that they examine higher-level knowledge structures, the feature scripts, to select concepts to be sent to CGEN. The knowledge structure level decision process is implemented in CADHELP by a module called HELPCON, programmed in Franz LISP [Fode80]. An entire feature script is input to HELPCON and it is responsible for traversing the links of the feature script and selecting concepts for expression by CGEN. The traversal of the script provides the main control for HELPCON, and at each link, HELPCON applies a particular rule which decides whether or not to express the link and the concepts it connects. There is one rule per link (see Figure 3.2) making HELPCON data-driven and easily expandable.

HELPCONs rules use several types of information to decide whether to express a concept or not. One is the type of link. A rule that fires because a certain link is present may do nothing more than

suppress expression of the link and the concepts that it connects. For example, CC is a rule that looks at causal couplings of states. In this domain, causal coupling can be inferred by the user, e.g. if the stylus is in a new location, the tip of the stylus, which is part of the stylus, is in a new location also. This type of information is domain-specific. In domains where the causal coupling of states may be less transparent, the CC rule could be reformulated to explain the coupling the first time it was encountered, then expect the user to be able to infer it.

Other link types that merely suppress the link and its concepts are THEN, IR, REASON, INITIATE. Since HELPCON is only concerned with the overt physical actions of the user, any mental actions or states are ignored in the explanation. Links connecting mental actions and states, whether those actions were by the system or the user, would be important if CADHELP was programmed to attempt to describe in detail user mistakes, or teach the user how to design, or even to debug feature scripts. For example, something like:

> I thought that the prompt would cause (initiate) you to decide to delete a particular device and that would be the reason you would move the stylus.

could be generated to explain why the system had waited for the stylus to be moved, when the user wasn't expecting to have to move it. Like the rules for mental links, the THEN rule also suppresses expression of the THEN link and the concepts it connects, since it does not really have enough information to decide if a concept should be said or not.

Two rules corresponding to links used in the CADHELP feature scripts are responsible for selecting important user actions and for focusing upon important states. These are OSE and OSC, respectively. The OSC rule operates upon one-shot-causal relationships, where an action causes a state. The action may be performed by the system or the user, and if it is performed by the system it and the resulting state are ignored. If the action is performed by the user, and it is an overt physical action (i.e. not a mental event) then OSC decides that this is something the user should be told. The state one-shot-caused by the action is not expressed, since the user is assumed to be able to infer the consequences of his actions. In a more complicated HELP situation, an explainer may want to tell the user the consequences of his actions, especially if they are in error.

The OSE link mainly serves to call attention to states the system expects the user will notice. In CADHELP, these are events like prompts appearing, objects blinking or devices appearing on the graphics display. Since the script describes the expected behavior of the user, these important states are represented as the object of user MTRANSs (mental transfers). Basically, OSE will select a state to be expressed if it sees that the state one-shot-enables the user to mtrans that state. HELPCON selects the state, rather than the user mtrans of the state, to avoid constructs like:

You will see a prompt appear on the screen.

The representation of these MTRANSs explicitly is useful in pinpointing important events to be animated by the animator, and could prove useful in explaining how to detect potential errors, for example:

If you do not see the prompt appear,
press the button or turn up the
intensity of the graphics device.

Another set of rules HELPCON uses aid in the traversal of the feature script. TRNPT is one of these. Feature scripts are organized temporally, but are not necessarily linear. At certain points in a script, there may be mutually exclusive paths that can be followed. For example, in CADHELP, the user can lengthen a connection or shorten a connection during the CONNECT command, but not both simultaneously. TRNPT is responsible for assuring that, when one of these turning points is reached, each path is traversed in turn. The user is assumed to know about the exclusiveness of the different paths, and no introduction like: "Do one of the following" is used.

An important property of parts of a feature script is that they can be repeated any number of times until some termination condition is reached, called RUT, for Repeat Until Threshold. This is useful for expressing segments of a script that are performed by the user in an incremental fashion until some desired state of the design is reached. For example, drawing a connection between two devices can be thought of as the process of drawing connected horizontal and vertical segments until the connection is complete. RUTs may be embedded, for example, each segment is the sum of many movements in a straight line. A RUT is defined by a satisfaction condition, which expresses the state that will cause the RUT to terminate, as well as a set of causally linked states and actions that are to be repeated. RUTs are handled by expressing the satisfaction condition embedded in the

construct: "Repeat the following until...", then subjecting the actions and states to be repeated to the HELPCON process.

Another property of feature scripts is that they can share large portions of standardized actions, e.g. moving a device using the stylus, using embedded scripts. The rule SR is responsible for deciding what to do with these scripts. The feature scripts used by CADHELP are represented so that the actual expansion of the embedded script is not inserted when it is called, but a pointer to an instantiated version is established. In the main path of the outer feature script is inserted a reference to the script, along with values for some script variables. This reference is causally linked to one or more states via SR, scriptal result. These states are the important conditions that are true in the world after the embedded script is executed.

The first time a reference to a script appears, HELPCONs SR rule expands it, i.e. it places all the actions and states making up that script into the mainstream of processing, where they are traversed. However, SR is sensitive to the number of times it has expanded an embedded script, so when it reaches subsequent references, it will say the intent of the script, then the main concept of the script. The intent of the script is goal concept, or the reason it is used, for example, one uses the $PRESS script to inform the system of some event in the design process. The main concept of the script is the (usually) single action that summarizes what the user is to do, in the CADHELP domain, it is usually summarized best by one verb and one

instrument, e.g "Press the stylus on the tablet", or: "Move the device with the stylus." Like SR, several of HELPCONs other rules are sensitive to previous explanations, namely OSE, OSC and RUT. The successively simpler explanations this process produces can be found in section 2.1.

HELPCONs rules use several kinds of information. Some use general knowledge of the users ability to make inferences and to remember what he has been told. Knowledge of the types of links that can connect concepts as well as the nature of the concepts in the script, is also used. In addition, keeping track of what has been said in a explanation aids in making subsequent explanations brief and to the point. Interesting extensions to this system are also possible, for no information relevant to describing the execution of a graphical feature has been thrown away in the feature script representation. To clarify this process, a small segment of a feature script going through HELPCON's pruning is shown in the next section.

3.4 A Detailed Example

The script to be examined by HELPCON is the embedded script, $move. This script is called from a main script by the following two concepts, linked by an SR:

```
($move actor *user              ~ the call to the move script,
       drag-obj *cursor          ~ where the object to be
       obj *stylus               ~ moved is the cursor.
       to &new-dev-srloc         ~ it is to be moved until it
       loc &newcurs-srloc        ~ overlaps the device the user
       sat-cond (p-config con1 *cursor  ~ has chosen to create.
                          con2 &desr-cr-dev
                          confrel (overlaps)
                          mode (nil)
                          time (nil)))
       ---- SR ----
(p-config con1 *cursor          ~ the important result after
          con2 &desr-cr-dev     ~ doing the script
          confrel(overlaps) mode(nil))
```

The script variables for $move are listed below. The call as shown above will instantiate these with the appropriate fillers. Notice that the stylus is not a script variable, since moving the stylus is the only way a graphical object can be moved in the CAD Tool.

Variables

(1) @drag-obj: the object that is to be moved, in this case, the cursor.

(2) @sat-cond: the satisfaction condition for the repeat until threshold

(3) @loc: the location that is going to be changing, in this case, the location of the cursor.

(4) @toloc: the location to which the moved object is going, used to decide if the RUT is finished.

The main concept and the intent of the script are also accessible to SR, in case it chooses a less verbose expression for the script. In this example, the script $move will be expanded, but the intent and main concept for $move are shown below in their uninstantiated form.

The main concept for $move: "Move some object with the stylus"

```
(ptrans actor *user
        obj @drag-obj
        to (nil) from (nil)
        via (nil) mode (nil) time (nil)
        inst (ptrans actor *user
                     obj *stylus
                     to (nil) from (nil)
                     via (nil) inst (nil)
                     mode (nil) time (nil)))
```

The intent concept for $move: "To move some object, use the stylus

```
(cause precon (s-goal actor *user
                      mode (nil) time (nil)
                      goal (ptrans actor *user
                                   obj @drag-obj
                                   to (nil) from (nil)
                                   mode (nil) inst (nil)
                                   focus (actor) via (nil)
                                   time (nil)))
       postcon (ptrans actor *user
                       obj *stylus
                       to (nil) from (nil) mode(nil)
                       inst (nil) focus (actor) via (nil)
                       time (nil)))
```

What follows is the actual expansion of $move, assuming it was called as above. The script is instantiated (by substituting in for each of the script variables) and each link will cause the appropriate rule in HELPCON to fire. The numbered comments indicate what is happening to the immediately preceding and following concepts at each step. In addition, an English version of each Conceptual Dependency representation precedes each concept and is indicated by a tilde, "~".

- The user maps the location of the device (which is where he wants to move the cursor to) to a location called &endpnt, which is the desired ending point.

```
($makemap actor *user
          con1 &new-dev-srloc con2 &endpnt
          maprel(corresp))
```

---- SR ----

(1) The SR rule disregards both of these, since they are performed by the system. This level of detail is needed so the graphical animator can mimic what the user is doing.

- The result of the mapping is an abstract configuration, i.e., the user knows that the &new-dev-srloc is the desired ending location of the cursor.

```
(a-config con1 &new-dev-srloc con2 &endpnt
          confrel(corresp))
```

---- OSE ----

(2) The OSE rule ignores any user mental action.

- This enables the user to decides where he is going to move the stylus.

```
(mbuild actor *user mobj &dltaloc)
```

---- REASON ----

(3) The REASON rule ignores the mbuild and the mknow, but the animator has now presumably picked out a destination point and knows it.

- The decision above is the reason he knows where the stylus is going to be moved.

```
(mknow actor *user mobj &dltaloc)
```

---- RUT ----

(4) This link begins the RUT. The RUT rule expresses the satisfaction condition of the RUT, prefaced by the introduction ("Do the following until the cursor is over the device you want to add to the design.")

- This is the beginning of the RUT. It points to all the following concepts, which are embedded in the RUT.

```
(rut sat-cond (p-config con1 *cursor
                        con2 &desr-cr-dev
                        confrel(overlaps) mode(nil))
```

----- INITIATE ----

(5) This rule allows HELPCON to ignore mental events that initiate user actions.

- The fact that the user knows where to move the stylus initiates him to move the stylus. This is also the beginning of the RUT.

```
(ptrans actor *user obj *stylus
        to &dltaloc from &sty-daloc
        via (nil) inst (nil) mode (nil) time (nil))
```

---- OSC ----

(6) The OSC rule notices that the user has performed some physical action resulting in some state, so it causes the action to be expressed ("Move the stylus to a new location") but ignores the state resulting from the action.

- The movement of the stylus has caused the location of the stylus to change.

```
(s-change actor *stylus
          mode (nil) time (nil)
          attr (loc val &newsty-daloc dir (to)))
```

---- OSE ----

(7) The OSE rule sees that the state is enabling a system action, not a user one, so it deems the state and the action unimportant, and neither is expressed.

- The change enables the system to notice the change in location of the stylus.

```
(mtrans actor *sys
        nobj (s-change actor *stylus
                       mode (nil) time (nil)
                       attr (loc val &newsty-daloc dir (to)))
        from (nil) to (*cp* part *sys)
        mode (nil) time (nil) inst (nil))
```

---- INITIATE ----

(8) The INITIATE rule ignores both the system perception and the mental action it initiated.

- The perception event on the part of the system initiates it to realize that the user has the goal of moving the cursor.

```
(mbuild actor *sys mode (nil) time (nil)
        mobj (s-goal actor *user mode (nil) time (nil)
                     goal ($draw actor *sys obj *cursor
                                 loc &newcurs-srloc
                                 mode (nil) time (nil)))
```

---- REASON ----

(9) The REASON rule ignores the system's reason and actions.

- The system's realization is the reason that the system attempts to form a correspondence between the location of the stylus on the tablet, and a location for the cursor on the screen

```
($makemap actor *sys
          con1 &sty-daloc con2 &newcurs-srloc
          maprel (corresp)
          mode (nil) time (nil))
```

---- SR ----

(10) The SR ignores the result of the embedded non-expandable script.

- The result of the makemap script is an abstract configuration between the location of the stylus on the tablet and a point on the graphics display.

```
(a-config con1 &sty-daloc con2 &newcurs-srloc
          confrel (corresp)
          mode (nil) time (nil))
```

---- CSE ----

(11) CSE will ignore any system actions.

- Knowing the point on the graphics display enables the system to draw the cursor at that point.

```
($draw actor *sys obj *cursor
       loc &newcurs-srloc mode (nil) time (nil))
```

---- SR ----

(12) The result of the execution of the $draw script is unimportant, from the point of view of SR.

- The result of the redrawing is a new location for the cursor.

```
(s-change actor *cursor
          attr (loc val &newcurs-srloc dir (to))
          mode (nil) time (nil))
```

---- OSE ----

(13) Since the state occurring is explicilty see by the user, it is deemed important, and the state is selected by OSE for expression ("The cursor moves to a new location"). The actual perception of the event by the user is ignored.

- The change in location of the cursor enables the user to detect the change.

```
(mtrans actor *user
        mobj (s-change actor *cursor
                       attr (loc val &newcurs-srloc dir (to))
                       mode (nil) time (nil))
        to (*cp* part *user) from (nil) inst (nil)
        mode (nil) time (nil))
```

---- INITIATE / REASON ----

(14) INITIATE/REASON ignores user mental actions/states.

- The user perception presumably led to an mbuild which then led to this mknow, but the initiate/reason link has allowed us to skip all that. Here the user either 1) knows that the cursor and the device he has decided to create overlap (since their locations overlap) and the $move script ends, or the two do not overlap, so the satisfaction condition is not met, and the RUT begins again.

```
(mknow actor *user
       mobj (p-config con1 *cursor
                      con2 &desr-cr-dev
                      confrel (overlaps)
                      mode (nil) time (nil)))
```

CHAPTER 4

Explanations in Academic Counseling

4.1 Introduction

The experience with CADHELP, as well as an examination of an actual explanation of the Computer Science curriculum at The University of Connecticut, suggested several improvements which a next-generation explainer should incorporate. Several of these improvements have to do with shortcomings that are a consequence of the domain used. CADHELP's explanations describe to an actor how to act to effect a computer-aided design. The user is assumed to have one goal, to change the state of the design. In domains where the actor is dealing with the social world also, the goals are more complex, one goal may lead to a series of subgoals, and plans for realizing those goals must be explained. These goal episodes can be used, however, to drive the explanation process, much as they are used to aid in understanding [Wile81].

A feature that wasn't critical for the domain in CADHELP, but is essential for brevity in more complicated domains is the use of examples. In the explanation studied the explainer would frequently invent a person with some set of characteristics in order to focus on a critical combination of traits, and then refer to this hypothetical person during the following sentences. An explainer utilizing such a feature would need to know when an example should be used, and how to

create an entity which focuses on the desired characteristics.

There is an aspect of explanations that the CAD domain simply did not allow us to investigate, the previewing and reviewing summarization process. Previews and reviews are used to provide a framework for the listener for what is to follow and to capture the important points of the previous discussion, respectively. CADHELP does implement a primitive form of previewing, when it explains the object or goal of the feature being explained first. However, a goal-oriented explanation utilizes these summarizations frequently. In the sample explanation the explainer previewed five paragraphs of information by saying: "You have to learn some programming," and reviewed at the end by stating: "So, at the end of this, you will have learned your introductory programming." This explanation technique arises naturally in an explanation that is goal driven, since the preview and review statements can be thought of as summaries of goals that are either about to be explained, or have just been explained.

CADHELP's model of explanations allows the user no chance to interrupt the explanation and ask questions in a mixed-initiative fashion (as in [Clan79], [Coll75]). This may become a problem since CADHELP's assumes that the user remembers everything he is told, and so becomes more and more laconic as the explanations continue. This feature may be essential to allow a system to deal with very naive users. Moreover, in addition to assisting the user, recording and examining the questions asked provides a good measure of the clarity of the explanations produced.

The phenomena described above must be incorporated into any process that attempts to perform complicated explanations. In order to provide the user with a framework of cohesive text, previewing, and reviewing must be performed. In addition to these, the explainer can economically focus on an object's desired characteristics using examples. Finally, if the explanation is to be flexible enough for a computer-naive user, the process must allow interruptions. These features are part of a model of the explanation process designed for use in the system described below.

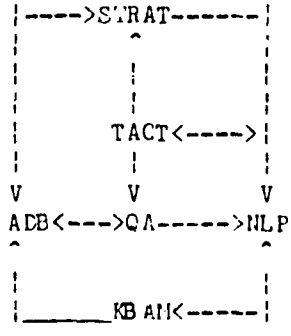
4.2 Overview of the Academic Counselor

A system is being developed called ACE (Academic Counseling Experiment) for research in conversational interaction and knowledge acquisition [Cull82]. ACE models an academic counselor who performs various tasks for a student such as conducting a preregistration or answering questions about courses. Part of the design of ACE was motivated by previous work in conversational systems. The GUS system of Bobrow et. al., [Bobr77], performs a constrained conversational task, which is mixed-initiative but attempts to retain control of the conversation. In GUS, the focus of control is on an attempt to fill out a frame for a user making an airplane trip. In a similar fashion, the conversational system for ACE focuses on filling out a next semester course schedule for a student, but its design is more robust, e.g., it allows the user to ask questions during the conversation.

ACE was designed to be an evolving system, and one that would be worked on by several persons. It consists of roughly 6 experts as

shown in Figure 4.1. The module labeled KBAM (Knowledge Base Acquisition Mechanism) is responsible for the learning part of ACE. Expert users explain facts about courses and scheduling to KBAM, which updates and modifies the knowledge in ADB (academic database). The ADB contains facts about particular students, the curriculum, and rules for distributing courses over four years of study, and contains a deductive retriever modeled after [Char80].

The part of the system responsible for conducting a preregistration for a student is labeled STRAT, for strategist. The strategist interacts with the user through the NLP module, which contains APE (A Parsing Expert) [Cull80], and CGEN, the conceptual generator. STRAT contains the conversational control for the task of filling out a next-semester schedule for a student. It asks the student questions (e.g. "Who are you?", "What courses are you taking?") and the answers,



Organization of the Academic Counselor

Figure 4.1

as parsed by APE, are filtered through the TACT (tactician) module. The tactician is responsible for seeing that the information requested by STRAT is in fact present in either the student's answer, or the past history of the interaction. Designing this tactician as a separate module was motivated by the observation that people normally give too little or too much information in response to questions.

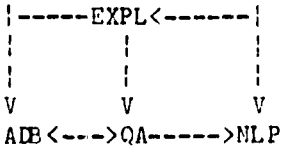
TACT may also discover that the user, rather than answering a question, has asked one of his own. Questions from the student are passed directly to the module Q/A. Q/A queries the ADB for answers to the student's questions, and gives the response through the NLP interface. Q/A is designed to answer simple slot-filler type questions about such things as who is teaching a course and prerequisite information. After the Q/A module answers the user's question, however, STRAT regains control and attempts to elicit more information from the student. When STRAT has sufficient information about who the student is and what courses he has taken in the past (or is currently taking), it requests that the ADB use it's knowledge of course requirements and scheduling to propose the next semester schedule. This ends the preregistration cycle.

4.3 The Explainer for ACE

Unlike the STRAT module controlling the preregistration task, the Explainer is not attempting to obtain information, but is attempting to give information. For this reason, the control for the explainer should be more flexible. The model for the explainer for ACE produces explanations by describing a set of goals, and rules for achieving

those goals. Some of the rules the explainer uses can be shared between it and STRAT. This explainer differs from other rule-based system explainers (e.g. [Shor76]) in that it does not explain those rules that have fired, but explains those rules that could fire. For this reason, it can be operated independently of the actual system. Indeed, one could imagine hooking up this explainer to any rule-based system and then asking it: "What do you know?" This is similar to Swartout's approach in explaining rules that could potentially fire to describe a system [Swar77].

The Explainer for ACE fits into the system as shown in Figure 4.2. It functions in ACE like the modules STRAT and TACT, i.e. as a controller. The Explainer interfaces directly with the user via the NLP module. Questions asked by the user that cannot be directly answered by the Explainer on the basis of previous context are sent to the Q/A module. The ADB is used by the Explainer to provide the specifics of the explanation of a particular curriculum.



Organization of the Academic Counselor with Explainer

Figure 4.2

The Explainer generates a series of goals represented in CD form. Goals can be realized by subgoaling. Rules for subgoaling exists as rules for goal realizations. The top-level goal is: "Explain how to get a bachelor's degree in EECS at UConn." This goal is broken into a series of subgoals, which, performed sequentially, realize the top-level goal.

As an example of this, suppose that the system has the goal of explaining the group distribution requirements. A rule for realizing this goal will cause the system to have goals for:

1) Explaining the Group 1 requirement

2) Explaining the Group 2 requirement

3) Explaining the Group 3 requirement

These three explanations can then be done directly. This will produce a sequence like the following.

```
You have to fulfill some distribution requirements.
There are three distribution groups.
Group one is the technical course group.
   You need 18 credits from these.
Group two is the Social Science group.
   You need 9 credits from these.
Group three is the Fine Arts group.
   You need 9 credits from these.
Those are the group distribution requirements.
```

Note that before realizing the goal of telling the user about the distribution requirements, the system previews the topic to come by stating the object of the the goal. The system remembers what goals it is trying to realize, and at the end of the sub-explanation, reviews its goals.

The explainer generates examples when the situation requires reference to a person meeting some special requirements. Otherwise, the user is addressed directly, as in the example above. Since the domain of ACE does not have many dissimilar entities (in fact, there are only a few) and courses and teachers are pretty much static, most example generation is done on students. A student progressing through four years of college is the focus, and the examples needed are students at given times in this four years, perhaps with other special requirements. The initial idea used in the explainer for ACE is to have a module that tracks all known entities and produces a known entity or makes one up as requested.

The most difficult portion to model is allowing interruptions by the user. This is partly due to the fact that little is known about the kinds of questions a student using ACE may ask. However, there are two general types that could occur. One is the fill-in-the-slot type question which can be identified by the question focus (usually what type questions). Questions of this form can be handled in the same manner as TACT handled them, by passing them off to the Q/A module. Another other type of questions are the why questions, which ask the explainer why it said (or omitted) certain information. Questions asking about things that have been said can be answered by tracing up the goal stack, and determining what caused certain rules to fire. Some why questions, however, ask about material the explainer has yet to describe. The overhead involved in tracing out possible paths in the large number of goals and rules ACE has is high and for now, these are best ignored.

This chapter concludes with a sample explanation this new model explainer could produce. The processes of previewing, reviewing, question answering and example generation are outlined in enough detail to demonstrate that the model is feasible to implement.

4.4 A Detailed Example

Below is an explanation fragment describing a typical technical course schedule for a freshman/sophomore.

Explainer:

(1) This schedule is a lower division schedule for technical courses.

(2) A first semester freshman will take Computer Science 110, Math 133 and Chemistry 127.

(3) A second semester student will take Computer Science 111, Math 134 and Chemistry 128.

Student:

(4) Why can't I take a Physics course the first semester?

Explainer:

(5) To take Physics 151 you must know Differential Equations.

(6) Differential Equations are taught in Math 200.

(7) Math 133 and Math 134 are prerequisite for Math 200.

This example illustrates the use of previewing, and response generation for questions. The preview on line 1 describes the intent of the explanation and is performed since a goal track is beginning. The student's question is answered by describing the rules the scheduler

used to determine the correct schedule. Lines 5 through 7 describe the prerequisite information contained in the curriculum structure which contributed to the choice.

A fairly detailed explanation of the process by which the above fragment may be produced is in order. While this has not been implemented, the following should give some idea of how such an implementation should proceed. The Explainer is currently attempting to fulfill the goal of explaining the EECS curriculum to a new undergraduate. One of the subgoals for fulfilling this is to explain the lower division (freshman and sophmore years) courses. The rule for this explanation will lead to two new goals, to explain the nontechnical and technical courses. Thus, at some point, a rule fires that looks like the following:

(1)

```
(explain-courses type (technical) sem (lower))
```

This will match against the following rule, with the variables indicated by '&'.

(2)

```
(to con1 (explain-courses type &type sem &sem)
    con2 (and con1 (explain-content
                        course-type &type
                        semester &sem)
                con2 (explain-schedule
                        course-type &type
                        semester &sem)))
```

Rule two states that in order to explain any set of courses, you must fulfill the goals of explaining their content and when they fit into the schedule. If we assume that the explanation of their content

has occurred, we arrive at the point where the fragment under study begins. First, the Explainer must perform a preview of what it is going to do next, since it has just finished the somewhat unrelated task of telling what information each course teaches, and is starting the next part of 2. So, sentence 1 is generated from the CD representation shown in 3.

```
(3)
 "This schedule is a lower division schedule for technical courses."
(a-config
    confrel (equiv)
    con1 (infostruc itype (schedule)
                          course-type (technical)
                          semester (lower)
                          course1 (nil)
                          course2 (nil)
                          course3 (nil)
                          course4 (nil)
                          ref (imm))
         con2 (infostruc itype (schedule)
                          course-type (technical)
                          semester (lower)
                          course1 (nil)
                          course2 (nil)
                          course3 (nil)
                          course4 (nil)
                          ref (indef)))
```

The Explainer then proceeds to explain a proposed schedule for the courses. The rule that tells it how to do this is as follows.

```
(4)
(to con1 (explain-schedule course-type &type
                           semester &sem)
      con2 (and (replace
                   str0 (gen-student
                           semester-standing (1)
                           background (nil)))
                (replace
                   str00 (explain (propose-sched
                                   course-type &type
                                   student str0)))
                (replace
                   str1 (gen-student
                           semester-standing (2)
                           background str00))
                (replace
                   str01 (explain (propose-sched
                                   course-type &type
                                   student str1)))
                (replace
                   str2 (gen-student
                           semester-standing (3)
                           background str01))
                (replace
                   str02 (explain (propose-sched
                                   course-type &type
                                   student str2)))
                (replace
                   str3 (gen-student
                           semester-standing (4)
                           background str02))
                (replace
                   str03 (explain (propose-sched
                                   course-type &type
                                   student str3)))))
```

In order to propose the schedule for a student at any given time, the scheduler is called, via 'propose-sched'. The only information the scheduler needs is the student's background and semester standing. A typical first, second, etc. semester student is generated by the call to 'gen-student', and is given more background each time. This is done so that successive applications of the rule 'propose-sched' generate a schedule for a new student. The results of the propose-sched (which becomes a call to the scheduling expert, ADB) are then shipped

to explain, which calls CGEN with the concept shown in 5.

(5)
"A first semester freshman will take Computer Science 110, Math 133 and Chemistry 127."

```
(simul con1 ($course actor (person persname (nil)
                                   surname (nil)
                                   ref (indef) convrole (nil)
                                   eprole (&fresh))
                           student &fresh
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks)
                                  cno (110)
                                  dept (org orgname (CS)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                           mode (modes mode1 (:t))
                           time (times time1 (:futr)))
            con2 ($course actor &fresh
                           student &fresh
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks) cno (133)
                                  dept (org orgname (Math)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                           mode (modes mode1 (:t))
                           time (times time1 (:futr)))
            con3 ($course actor &fresh
                           student &fresh
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks) cno (127)
                                  dept (org orgname (Chem)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                          mode (modes mode1 (:t))
                          time (times time1 (:futr)))
            compnum (3))
```

The concept shown in five represents the simultaneous occurrence of three instances of the execution of the $course script. Each of the instances of $course has an actor, student and teacher. If the actor and the student are the same, the generator uses take, otherwise it

the actor and the teacher are the same it uses teach. The result of this generation is sentence 2. The concept &fresh is used as a shorthand in the representation, it should be thought of as being replaced with the first actor (i.e. &fresh really means (person persname ...) Sentence 3 is generated from a similar representation, shown in 6.

(6)
"A second semester student will take Computer Science 111, Math 134 and Chemistry 128."

```
(simul con1 ($course actor (person persname (nil)
                                   surname (nil)
                                   ref (def) convrole (nil)
                                   eprole (&soph))
                           student &soph
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks) cno (111)
                                  dept (org orgname (CS)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                           mode (modes mode1 (:t))
                           time (times time1 (:futr)))
            con2 ($course actor &soph
                           student &soph
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks) cno (134)
                                  dept (org orgname (Math)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                           mode (modes mode1 (:t))
                           time (times time1 (:futr)))
            con3 ($course actor &soph
                           student &soph
                           teacher (nil)
                           obj (infostruc
                                  itype (acad-ks) cno (128)
                                  dept (org orgname (Chem)
                                            orgtype (acad-dept)
                                            orgocc ($course)))
                           mode (modes mode1 (:t))
                           time (times time1 (:futr)))
           compnum (3))
```

After Sentence 3 has been generated, the listener interrupts with a

question. The question is parsed by APE, and results in the concept shown in 7.

```
(7)
"Why can't I take a Physics course the first semester?"
(cause precon (:q)
       postcon ($course
                    actor (person persname (nil)
                                  surname (nil)
                                  ref (def)
                                  convrole (*other)
                                  eprole (nil))
                    student (person persname (nil)
                                    surname (nil)
                                    ref (def)
                                    convrole (*other)
                                   eprole (nil))
                    teacher (nil)
                    obj (infostruc
                           itype (acad-ks) cno (nil)
                           dept (org orgname (Physics)
                                     orgtype (acad-dept)
                                     orgocc ($course)))
                                     ref (indef)))
                     mode (modes mode1 (:neg) mode2 (:pntnt))
                     time (times time1 (:pres))
                     abstime (dur durtype (sem)
                                  val (1))))
```

Because the question refers to the antecedent of a cause, the Explainer must find a case where such a cause exists. This is in the rules used by the ADB. The Explainer first examines the rules used by the ADB to propose the schedule for the first and second semesters, to see if a Physics course was proposed and eliminated for a reason which could be explained. It happens that Physics 151 was a candidate, but was thrown out by the curriculum knowledge. This refers to the fact that since Physics 151 teaches Introductory Physics, knowledge of Differential Equations is necessary before taking it. The Explainer then

examines the ADB curriculum knowledge to see where Differential Equations are taught. This turns out to be Math 200. Since the question is answered, the Explainer could stop, but it checks to see if it's mentioning anything new. Physics 151 is new, but is ruled out since the student had mentioned it in the question. Math 200 is new, so the Explainer checks its prerequisites, and tells the student what they are. So, this first reason becomes the three concepts shown in 8 through 10.

```
(8)
"To take Physics 151 you must know Differential Equations."
(cause precon
        (s-goal actor *other
                mode (modes mode1 (:t))
                time (times time1 (:pres))
                goal ($course actor *other
                             stud   *other
                             teac..r (nil)
                             obj (infostruc
                                    ity pe (acad-ks)
                                    cno (151)
                                    dept (org orgname (Physics)
                                              orgty pe (acad-dept)
                                              orgocc ($course))
                                    ref (def))
                             mode (modes mode1 (:t))
                             time (times time1 (:pres))))
      postcon (mknow actor *other
                     mobj (infostruc
                             ity pe (acad-ks)
                             subj (ks ty pe (Math)
                             value (DiffEq))
                             cno (nil)
                             dept (org orgname (Math)
                                       orgty pe (acad-dept)
                                       orgocc ($course))
                             ref (def))
                     time (times time1 (:pres))
                     mode (modes mode1 (:t) mode2 (:oblig))))
```

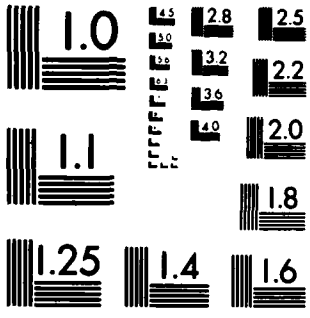
```
(8)
"Differential Equations are taught in Math 200."
(cause
  precon (s-goal
            actor *other
            mode (modes mode1 (:t)) time (times time1 (:pres))
             goal ($course
                     actor *other
                     student *other
                     teacher (nil)
                     obj (infostruc
                            itype (acad-ks) cno (200)
                            dept (org orgname (Math)
                                      orgtype (acad-dept)
                                      orgocc ($course))
                            ref (def))
                     mode (modes mode1 (:tf))
                     time (times time1 (:pres))))
  postcon (mknow actor *other
                 mobj (infostruc itype (acad-ks)
                                 subj (ks type (Math)
                                          value (DiffEq))
                                 cno (nil)
                                 dept (org
                                        orgname (Math)
                                        orgtype (acad-dept)
                                        orgocc ($course))

                                 ref (def))
                 time (times time1 (:pres))
                 mode (modes mode1 (:t) mode2 (:oblig))))
```

AD-A126 190 GENERATING NATURAL LANGUAGE EXPLANATIONS IN A COMPUTER-AIDED DESIGN SYSTE..(U) CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RESEARCH

2/2

UNCLASSIFIED M A BIENKOWSKI ET AL. JAN 83 TR-CS-83-1 F/G 9/2 NL

END
DATE FILMED
4-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
(9)
"Math 133 and Math 134 are prerequisite for Math 200."
(a-config
  confrel (prereq)
  con1 (simul
        compnum (2)
        con1 ($course
                    actor (person persname (nil)
                                  surname (nil)
                                  ref (def) convrole (nil)
                                  eprole (&student))
                    student &student
                    teacher (nil)
                    obj (infostruc itype (acad-ks) cno (133)
                                   dept (org orgname (Math)
                                             orgtype (acad-dept)
                                             orgocc ($course)))
                   mode (modes mode1 (:t))
                   time (times time1 (:pres)))
              con2 ($course
                    actor &student
                    student &student
                    teacher (nil)
                    obj (infostruc itype (acad-ks) cno (134)
                                   dept (org orgname (Math)
                                   orgtype (acad-dept)
                                   orgocc ($course)))
                    mode (modes mode1 (:t))
                    time (times time1 (:pres))))
  con2 ($course
        actor &student
        student &student
        teacher (nil)
        obj (infostruc itype (acad-ks) cno (200)
                       dept (org orgname (Math)
                                 orgtype (acad-dept)
                                 orgocc ($course)))
        mode (modes mode1 (:t))
        time (times time1 (:pres))))
```

If the course had failed for scheduling reasons (e.g. both Chemistry 127 and Computer Science 110 are lab courses and one can't take more than two lab courses per semester), the Explainer would have explained those reasons also.

CHAPTER 5

Conclusions

Language generation has been studied by many researchers to enhance the friendliness of computer systems. A useful way to study relatively complicated generation tasks is through explanations of the knowledge stored and used by a given program. This knowledge may be in any form, but the most flexible and natural-sounding text will be produced if the program has some idea of the meaning of any piece of the knowledge structure. In addition to the increased naturalness of the text, the explanation may be produced in a different modality than a language.

Explanations werer found to fall into three broad categories. The simplest was explanation of a physical process, the next category arose when the physical process was acted upon by an actor. The most complex explanations were those explaining the interactions of actors in social situations. This breakdown was mainly due to the different types of domains that could be explained. It was shown that the information a computer program would need to know could be represented in a usable form. This one done with both a set of goals and rules, and CD structures connected with causal and enablement links.

The generation process was implemented in this work with two basic modules. One is CGEN, a system which will generate an English sentence from a CD representation of that sentence. CGEN uses a CD

structure to select a word, then expresses those concepts not covered by the word. The concepts not covered are placed relative to the word chosen by syntatic predicates expressed using precedes and follows. In addition to the basic cycle of finding words, CGEN uses a set of sketchification rules to modify the concepts being expressed to produce natural-sounding text. The sketchifiers represent domain independent knowledge, and are added to CGEN as needed. The only domain specific knowledge CGEN has is about words and their meanings. New words are added to CGEN to enable it to handle new domains.

The other module is a concept selection mechanism (embodied in HELPCON or ACE's explainer) which applies a set of rules to a knowledge structure and generates the most economical means of expressing it. The rules used here are more domain specific than CGEN's, but for any explainer, certain explanation strategies can be found, such as "If you mention a new word or phrase, describe it". The knowledge these explainers examine is frequently used by other modules for other purposes, so they must be flexible. Making them data-driven is one way to accomplish this.

The first concept selector studied was designed to select concepts from a feature script for expression by CGEN. This module, called HELPCON, was part of the Explanation Mechanism of the Computer-Aided design system, CADHELP. HELPCON selected concepts to be used as prompts or as part of the text of an explanation, given with graphical animation. Each command CADHELP could execute was represented as a feature script, a sequence of concepts represented in

Conceptual Dependency, connected by causal links. HELPCON traversed a given feature script, choosing concepts based on, 1) what the user was assumed to know, 2) what the user had previously been told by HELPCON and 3) what the user could be assumed to be able to infer from his knowledge of the world and the causal mechanisms that operate in it. This type of selection could occur because HELPCON represented the meaning of a feature in the feature scripts.

The explanation process was further studied in the domain of an Academic Counseling program. A sample explanation was studied, and a module was designed which would fit in with the existing system, ACE, and explain the undergraduate EECS curriculum at The University of Connecticut. This module consisted of a set of goals, and means for achieving those goals (through more goals or rules). Some of the more common rules could be shared with the module conducting an interview to perform preregistration, STRAT. The Explainer for ACE was an improvement over HELPCON, since it performed previewing, reviewing, generated examples, and allowed for some interruptions. While this Explainer was not implemented in ACE, an example with enough detail to suggest an implementation was given.

This thesis investigated the important issue of natural language generation in two domains which were sufficiently dissimilar to demonstrate that 1) the conceptual generator, CGEN, was sufficiently general to be used with both systems, and 2) the process of concept selection is important enough to be studied separate from the generation component. It may turn out that the two types of decisions made

in each of these processes is of such a similar nature that they can be conveniently implemented as one, but designing them separately eases the task for the developer.

Bibliography

[Barr81] Barr, A. and Feigenbaum, E.A., Eds., The Handbook of Artificial Intelligence. Los Altos, CA: William Kaufman, Inc., 1981.

[Birn81] Birnbaum, L. and Selfridge, M., "Conceptual Analysis of Natural Language," In Inside Computer Understanding. Schank, R., and Riesbeck, C., Eds., Hillsdale, NJ: Lawrence Erlbaum, 1981.

[Bobr77] Bobrow, D. et. al., "GUS - A Frame-Driven Dialog System." Artificial Intelligence, Volume 8, 1977.

[Carb70] Carbonell, J. R., "AI in CAI: An Artificial Intelligence Approach to Computer-Aided Instruction," IEEE Transactions on Man-Machine Systems, Volume MMS-11, 1970.

[Char77] Charniak, E., "Ms Malaprop, A Language Comprehension Program," Proc. 5th International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

[Char80] Charniak, E., Riesbeck, C. and McDermott, D., Artificial Intelligence Programming. Hillsdale, NJ: Erlbaum Press, 1980.

[Ches76] Chester, D., "Translating Mathematical Proofs into English," Artificial Intelligence, Volume 6, 1976.

[Clan79] Clancey, W. J., "Dialogue Management for Rule-Based Tutorials," Proc. Sixth International Joint Conference on Artificial Intelligence, Tokoyo, 1979.

[Coll75] Collins, A., Warnock, E. H., and Passafiume, J. J., "Analysis and Synthesis of Tutorial Dialogues," Psychology of Learning and Motivation, Volume 9, 1979.

[Cull80] Cullingford, R.E., and Pazzani, M.J., "Word meaning selection in multiprocess language processing programs", EE&CS Department TR-80-12A, The University of Connecticut, Storrs, CT, 1980.

[Cull81a] Cullingford, R. 1981. "Script Application." In Schank, R., and Riesbeck, C. (eds.), Inside Computer Understanding. Erlbaum, Hillsdale, NJ.

[Cull81b] Cullingford, R. E., Krueger, M. W., Selfridge, M., and Bienkowski, M. A., "Towards Automating Explanations," Proc. 7th International Joint Conference on Artificial Intelligence, Vancouver, B.C, 1981.

[Cull81c] Cullingford, R. E., Krueger, M. W., Selfridge, M., and Bienkowski, M. A., "Automated Explanations as a Component of a CAD system," IEEE Transactions on Systems, Man and Cybernetics, Volume SMC-12, Number 2, 1982.

[Cull82] Cullingford, R.E., and the UConn Intelligent Systems Group. "Purposive Conversation with ACE: An Academic Counseling Experiment," IEEE Proc. of the International Group Conference on Cybernetics & Society, Seattle, WA, 1982.

[Dave74] Davey, A., Discourse Production, Edinburgh, UK: Edinburgh University Press, 1974.

[Fenc82] Fenchel, R. S., and Estrin, G., "Self-Describing System using Integral Help," IEEE Transactions on Systems, Man and Cybernetics, Volume SMC-12, Number 2, 1982.

[Fode80] Foderato, J., "The FRANZ LISP Manual." In Volume 2c of Documents for the Berkeley UNIX Time-Sharing System. Dept. of EE&CS, Univ. of California, Berkeley, 1980.

[Gold75] Goldman, N., "Conceptual Generation," in Conceptual Information Processing, R. Schank, Ed. New York, NY: North Holland, 1975.

[Haye79] Hayes, P. and Reddy, R., "Graceful Interaction in Man-Machine Communication," Proc. 6th International Joint Conference on Artificial Intelligence, Tokyo, Japan, 1979.

[Jose83] Joseph, L., "A Heuristically Optimal Knowledge Base Organization Technique", EE&CS Department TR-83-2, The University of Connecticut, Storrs, CT, 1983.

[Marc82] Marcus, R. S., "User Assistance in Bibliographic Retrieval Networks through a Computer Intermediary," IEEE Transactions on Systems, Man and Cybernetics, Volume SMC-12, Number 2, 1982.

[McDo81] McDonald, D. D., "Natural Language Generation as a Computational Problem: an Introduction," COINS Technical Report 81-33, University of Massachusetts at Amherst, Amherst, MA, 1981.

[McKe80] McKeown, K. R., "Generating Relevant Explanations: Natural Language Responses to Questions about Database Structures," Proceedings of the First Annual National Conference on Artificial Intelligence, AAAI-80, Stanford, CA, 1980.

[Meeh81] Meehan, J., "Tale-Spin," In Inside Computer Understanding. Schank, R., and Riesbeck, C., Eds., Hillsdale, NJ: Lawrence Erlbaum, 1981.

[Mins75] Minsky, M., "A Framework for Representing Knowledge," in The Psychology of Computer Vision. P. H. Winston, Ed. New York, NY: McGraw-Hill, 1975.

[Neim82] Neiman, D., "Graphical Animation from Knowledge," Proceedings of the National Conference on Artificial Intelligence, AAAI-82, Stanford, CA, 1982.

[Norm75] Norman, D.A. and Rumelhart, D.E., Explorations in Cognition. San Francisco, CA: W.H. Freeman & Co., 1975.

[Phel82] Phelps, D., "Help Protocols in a Self-Explanatory CAD System," EE&CS Department TR-82-10, The University of Connecticut, Storrs, CT, 1982.

[Reic78] Reichman, R., "Conversational Coherency," Cognitive Science, Volume 2, Number 4, 1978.

[Rieg77] Rieger, C. and Grinberg, M., "The Declarative Representation and Procedural Simulation of Causality in Physical Mechanisms," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

[Scha75] Schank, R. C., Ed., Conceptual Information Processing. New York, NY: North Holland, 1975.

[Scha77] Schank, R. and Abelson, R., Scripts, Plans, Goals and Understanding. Hillsdale, NJ: Lawrence Erlbaum, 1977.

[Shap75] Shapiro, S. C., "Generation as Parsing from a Network onto a Linear String," American Journal of Computational Linguistics, Microfiche 33:45, 1975.

[Shor76] Shortliffe, E.H., Computer-Based Medical Consultations: MYCIN, New York: Elsevier/North Holland.

[Simm72] Simmons, R. and Slocum, J., "Generating English Discourse from Semantic Networks," Communications of the ACM, Volume 15, Number 10, 1972.

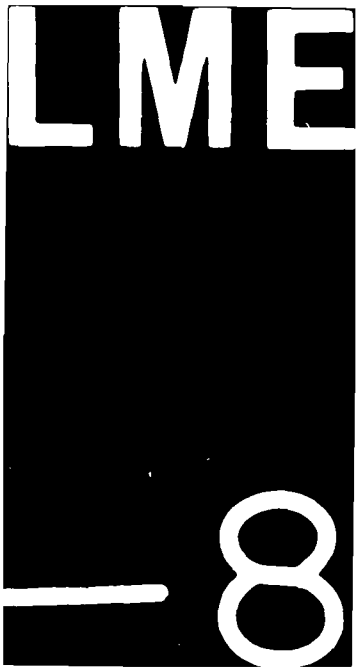
[Swar77] Swartout, W., "A Digitalis Therapy Advisor with Explanations." Proc. 5th International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977.

[Unge82] Unger, R., "Maintaining Context for Story Understanding," Unpublished Masters thesis, The University of Connecticut, Storrs, CT, 1982.

[Wile81] Wilensky, R., "PAM", In Inside Computer Understanding. Schank, R., and Riesbeck, C., Eds., Hillsdale, NJ: Lawrence Erlbaum, 1981.

[Wilk76] Wilks, Y., "A Preferential, Pattern-Seeking, Semantics for Natural Language Inference," Artificial Intelligence, Volume 6, 1976.

[Wino72] Winograd, T., Understanding Natural Language. New York, NY: Academic Press, 1972.


LME
8