MIT/LCS/TM-79

# A SYSTEM TO PROCESS DIALOGUE: A PROGRESS REPORT

GRETCHEN P. BROWN

OCTOBER 1976

MIT/LCS/TM-79

# A SYSTEM TO PROCESS DIALOGUE: A PROGRESS REPORT

Gretchen P. Brown

October 1976

MIT/LCS/TM-79

# A System to Process Dialogue: A Progress Report

Gretchen P. Brown

October 1976

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661.

Massachusetts Institute of Technology

Laboratory for Computer Science

(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

## Abstract

This is a progress report on work toward an English language interface for expert systems. A framework for handling mixed-initiative English dialogue in a console session environment is discussed, with special emphasis placed on recognition. The ideas presented here are being implemented in a prototype system called Susie Software, which is embedded in the OWL system. OWL is currently under development in the Automatic Programming Group at the M.I.T. Laboratory for Computer Science. We are using OWL to explore the problems of constructing expert systems, and for Susie Software the domain of expertise is programming. In the Susie effort to date, major emphasis has been placed on the construction of a computational model for the structural aspects of English dialogue; it is this structural model that will be discussed.

The research reported here was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661.

## Acknowledgements

I am grateful to William Martin and members of the Automatic Programming Group for helpful criticism and advice. The specifications for Susie Software were developed by Professor Martin, and many of the ideas presented here grew out of discussions with him. My thanks to Lowell Hawkinson, Rand Krumland, William Long, William Mark, Alexander Sunguroff, and Peter Szolovits for help with earlier versions of this paper.

A System to Process Dialogue: A Progress Report

Contents

## 1. Introduction: The Susie Software Environment

This paper is a progress report on the development of a framework for handling mixed-initiative English dialogue in a console session environment. The framework is being implemented in a prototype system called Susie Software.[1] We are using the Susie system to explore the interface between a user and an expert system, in this case an automatic programming system. The user is assumed to be knowledgeable in the subject area of the programs to be written but not necessarily knowledgeable about computer operations or computer languages. This user is, in short, the type of person that projects of diverse sorts -- management information, medical records, and so forth -- would like to reach if only there were a good "English language frontend." Lacking this facility, applications system designers have had to make do with high level command languages, automated questionnaires, or a limited subset of English. These solutions suffer to varying degrees from inflexibility and the need for users to translate their thought processes into the system's terms. Given a system with an English language facility, the user would have the flexibility of natural language, unencumbered by the need to learn and remember new languages or special rules.

This "given," of course, is the catch. The flexibility of natural language that makes it so attractive has led to processing problems of the type long since familiar to Artificial Intelligence researchers: either combinatoric explosion slows a system beyond usefulness or the reliability of reponses is sacrificed. This latter problem is potentially the more serious, since it can be less obvious to the user.

In the Susie Software effort, we hope to achieve a first level of flexibility without sacrificing reliability and while keeping response time within reasonable bounds. To do this, Susie's environment has been carefully constrained along the following dimensions:

1. communication channel
2. choice of application area
3. number of tasks performed
4. complexity of semantic domain

Looking at these dimensions in turn, communication in the Susie environment is limited to typed input and output, so that the special problems of speech processing are avoided. We are also assuming that input is free of spelling errors and gross grammatical inaccuracies (although full sentences are not required). Finally, there is a slight control bias on the side of the system, since the user may type only when an asterisk is output by Susie. This is a relatively minor limitation, merely ruling out mid-sentence interruptions by the user.

1. Susie Software is embedded in the OWL system, which runs on a PDP-10 under ITS. OWL is being developed at M.I.T's Laboratory for Computer Science by the Automatic Programming Group under the direction of William A. Martin. Development of the Susie Software dialogue facility began in 1974. An initial version of the dialogue OWL code was written in June 1974 and has been revised since that time.

The second constraint is the choice of application area. Here, the decision to construct a task-oriented environment (in the sense used in <Deutsch>) as opposed to, say, a system to model casual dialogue already acts as a constraint. For example, a task environment defines the aspects of user input that are important, so that the problem of deciding why a piece of information was input is greatly reduced. Beyond this, the structure of the primary task area chosen for Susie Software -- programming -- allows us to exploit strong expectations once a specific task is underway. At any given point in the course of producing a program there are only a small number of basic ways that the user and the system may interact (although the *content* of these interactions, of course, will vary considerably).

Third on our list of constraints is the number of different tasks performed. Susie's projected abilities are limited to writing programs and answering user questions about system capabilities and the programs that have been written. This means that the range of user requests for new tasks is relatively small. Since requests for new tasks occur at a time when expectations derivable from context are either weak or nonexistent, I suspect that this constraint will turn out to be an important one in keeping our first pass at a flexible system computationally manageable.

The fourth and final constraint is on the scope of the semantic domain. We have been working with the world of 2-dimensional toy blocks, a common starting point due to the simplicity of the domain and its clearly defined semantics. The ultimate goal for Susie Software is a business environment, which would make more realistic demands on the system. In either case, we feel that the semantic domain must be spanned in order for a system to be practical. By "spanned" we do not necessarily mean that every possible user question can be answered or every possible program produced. Instead, we mean that the system should have a good enough model of itself so that questions or requests that cannot be handled can be given appropriate responses. A simple "I don't understand" will not be considered adequate. Note that at present no attempt is made to span the semantics of the blocks world, and correspondingly the implementation lacks robustness. With many of the structural issues out of the way, we will be able to turn our attention to the semantic domain in the coming months. It is hoped, of course, that the groundwork that has been layed in the organization of concepts will hold us in good stead.

In addition to the four constraints -- communication channel, application area, number of tasks, and semantic complexity -- we also hope to avoid combinatoric problems by constructing a good model of the structural organization of dialogue. In a sense, this organization is the grammar of the dialogue, although the model that will be presented here is a computational one, not easily reduced to a set of rules. (No such attempt has, in fact, been made.) I use the term *structure* for what others might call the syntax of dialogue. Many "structural" phenomena are semantic in flavor (although they do not necessarily vary according to the specific semantic domain), and the use of the term *syntax* might be misleading. For example, the fact that questions get answers is easily enough called "dialogue syntactic," but the fact that answers may have associated stipulations ("Yes, if...") or qualifications ("Yes, but...") begins to stretch the connotations of the term *syntactic*. I will therefore call the model of dialogue to be discussed a structural one; the reader will get a clearer idea of the aspects of dialogue that are being labelled structural in the rest of the paper.

The Susie Software environment, then, has two properties that we consider crucial for our first pass at flexible, practical, man-machine dialogue. First, it is carefully constrained, and, second, the system contains a model of the structural aspects of dialogue. A sample of the sort of dialogue that could be expected in this environment is shown in figure 1. Here and in other examples of dialogue, asterisks precede lines input by the user. The dialogue is part of a longer one that has served as an implementation goal, since it presents a number of processing problems that are not handled in existing natural language systems. The dialogue in figure 1 is partially operational at this time, and it will be used in this paper to illustrate the natural language processing mechanisms that are being developed.

As mentioned above, the Susie Software system is embedded in OWL. Since a general understanding of the aims and operation of OWL is necessary for an understanding of the dialogue work, I will begin with a brief introduction to the OWL system and representation principles.

Hello, I am Susie. What is your name?
◦ Bill.
◦ I need a program to manipulate blocks written.

I can help you write a program which builds a scene using the blocks from another scene.
◦ That is what I want.

O.K. I need a description of the initial and final scenes.
◦ The final scene contains 3 blocks: B1, B2, and B3.

O.K.
◦ What else do you want to know about the final scene?

What are the positions of B1, B2, and B3 with respect to each other?
◦ B1 is on top of B2 and B3 is on top of B1.

O.K.
◦ Initially, all of them are on the table.

O.K.
◦ Can you write the program from that?

I have enough information to start writing the program. I can't determine all of the information which I will need until I write the program.
◦ O.K.

Do you want me to decide implementation issues such as what hand to grasp a block with?
◦ Yes, if I can ask you about them later?

You can.

I have finished the program. Do you want me to describe it?
◦ Yes.

Using my left hand, first I move B2, then I put B1 on top of B2, and then I put B3 on top of B1.
◦ How did you decide to use your left hand to move B2?

I needed to choose a hand with which to grasp B2. Both of my hands were free at that point. You had told me to decide the implementation issues. I chose the first hand which I thought of.
◦ Good-bye.

Figure 1.

## 2. An Introduction to OWL

The OWL system is continually evolving toward two goals: first, to provide an environment for the representation and use of expert knowledge and, second, to do limited-domain processing of natural language. It is our belief that the paths to meet these two goals are not completely disjoint and that some of the organizational principles and structures used to handle English will carry over to the structuring and use of expert knowledge as well. In this section, I will first survey the major modules of the version of OWL on which this work has been based, and then look at the notation used in that version. (The OWL version used became operational in September 1975; since that time, a new version of the system has been under construction.)

The major modules of the OWL system are shown in figure 2.[2]

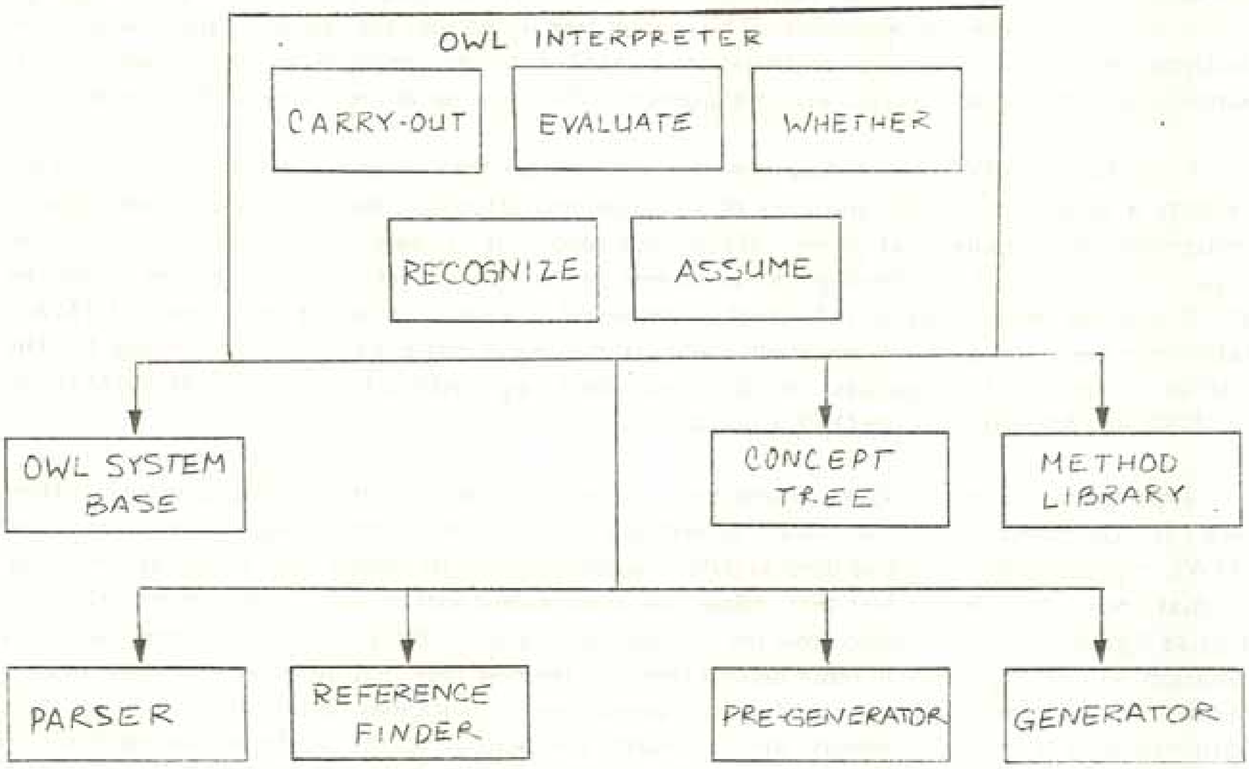

Figure 2.

2. The OWL system base was implemented by Lowell Hawkinson, and implementations of the interpreter's Carry-out, Evaluate, and Whether modules were done by Alexander Sunguroff, William Long, and William Swartout respectively. Peter Szolovits is responsible for the parser implementation, and the generator was done by William Swartout. Other modules are being developed by the author.

The OWL system base is described in <Hawkinson>, and is used to build and maintain the knowledge base. For our purposes here, it is sufficient to say that OWL representations are made up of data structures called *concepts*, using the operations of *specialization* and *modification*. Specialization is the subcategorization operation used for hierarchical ordering of concepts. Modification allows properties, including complex structures such as procedures, to be associated with concepts. Both the (immediate) specializations of a concept and its properties are found on its *reference list*. OWL concepts occurring in this paper will be represented using capital letters.

The backbone of the knowledge base is the concept tree, the September 1975 version of which was constructed by William Martin. The tree contains concepts for the words of Basic English (<Ogden>) plus other concepts of general applicability, among them a set of semantic cases. The concept tree is used by every module in the OWL system, although much of the original organizational impetus was the attempt to reflect regularities perceived in English usage. Each expert system embedded in OWL will bring its own set of concepts to add as specializations of the already-existing tree, and this augmented tree will then be used in both natural language processing and reasoning operations. The concept tree is one (but not the only) place where the analysis of natural language organization is applied to the problem of organizing expert knowledge. Individual concepts will be explained as they come up in the course of the discussion.

Note that the OWL system has opted for a tree rather than a more general hierarchy. This reflects a simplification for purposes of computational efficiency, but it does not constitute a restriction on computational power. While each concept is assumed to have only one *primary* superclass, other class memberships can be entered on the concept's reference list. For example, the OWL concept representing an individual person would be a specialization of the concept HUMAN (although not necessarily an immediate specialization), but it might have on its reference list the various roles and properties of the individual, e.g. ADULT, LISP-PROGRAMMER, VEGETARIAN, (AGENT (MOVE BLOCK-A)), etc.

This brings us to the OWL interpreter. The interpreter executes OWL structures called METHODs, which belong to OWL's incarnation as a very high level programming language. (OWL representations are also used as data structures, e.g., parser output, and in the encoding of factual knowledge.) In the text that follows, the word *method* will be used in this technical sense. Figure 3 gives a sample method from the semantic domain of toy blocks. OWL representation and notation will be explained in more detail below, but for now note that methods contain a header (as a unique name), input case specifications, output case specifications (optional), a series of steps (principally calls to other methods), and, if desired, assertions about the results of the method. In figure 3, the header is the OWL concept ((PUT ENTITY) ((ON TOP) ENTITY)), the input case specifications are for the cases OBJECT, AGENT, INSTRUMENT, and SPECIFIC-LOCATION, and the output case specification uses PRINCIPAL-RESULT. It is the job of the interpreter module Carry-out to go through the steps of a method, matching against calls to find appropriate subprocedures. The input case restrictions associated with a method are used in this matching process. Note that in the current OWL system no attempt is made to simulate distributed control, e.g., in the use of demons. Execution of methods, then, is highly centralized, and guided by a single control loop in the interpreter.

```
[((PUT ENTITY) ((ON TOP) ENTITY))
 PLAN
 OBJECT:             <--   ENTITY:1
 AGENT:              <--   (OR HUMAN: VERBALIZER:)
 INSTRUMENT:           <--   [HAND: (PART AGENT:) <-- ::]
 SPECIFIC-LOCATION:   <--   ((ON TOP) (ENTITY:2 THE))
 PRINCIPAL-RESULT:    <--   (LOCATION OBJECT: SPECIFIC-LOCATION:)

 METHOD: <--
  ;;;1
  (DISPLAY SCENE):1,

  ;;;2
  (FIND ([SPACE:
       LOCATION::         <--   SPECIFIC-LOCATION:
       ASSIGNMENT-FOR::   <--   OBJECT:]
         SOME)),

  ;;;3
  (GRASP OBJECT:),
  ;;;4
  (DISPLAY SCENE):2,

  ;;;5
  [(MOVE INSTRUMENT:)
   DESTINATION:: <--
      [PLACE:
       [LOCATION:::
         Y-COORDINATE:::: <-- (PLUS (Y-COORDINATE
                                 (LOCATION (SPACE: THE)))
                                (DIMENSION (HEIGHT OBJECT:)))
         X-COORDINATE:::: <-- (VALUE (X-COORDINATE
                                 (LOCATION (SPACE: THE))))]]],

  ;;;6
  (BECOME PRINCIPAL-RESULT:),
  ;;;7
  (DISPLAY SCENE):3,
  ;;;8
  ((LET GO) OBJECT:),
  ;;;9
  (DISPLAY SCENE):4]
```

Figure 3

A record of the execution process is kept in the *event tree*, which is also used by the interpreter in making control decisions. The events on the tree correspond to the substeps of the methods executed. Past events are not removed from the tree, so that they are available for inspection, question answering, resumption (in the case of uncompleted events), etc. It is important not to confuse the event tree with the concept tree. The former is built by the interpreter as a record of methods executed in the course of a console session, while the latter is a part of the knowledge base, embodying the first cut at organizing all of the concepts known to the system, not only the events.

Several interpreter modules come into play in the course of carrying out a method. The module Evaluate takes OWL forms and returns appropriate instantiations, for example when a call is evaluated before the search for a method to carry it out. The module Whether is responsible for reasoning, using a combination of built-in strategies and user-supplied procedures to determine whether or not a condition holds in the current operating environment. This module processes the antecedents of IF-THEN steps, which are the conditional steps in OWL. If difficulties arise in the course of carrying out a step, a series of failure mechanisms are available to the interpreter. The ultimate goal for the OWL failure handler is that it be flexible enough to take appropriate action in a wide range of situations. The existing failure mechanisms are ordered in terms of power, so that small problems receive local patches, while more powerful solutions are reserved for more extensive difficulties. Central to the OWL approach is the idea of a powerful analysis facility that will be able, in those cases where it is necessary, to allow the interpreter to "back off" and reconsider its choice of strategies. Work toward this goal is ongoing, and we feel that the knowledge base organization and the information structuring as well as the control flexibility of the event tree provide a foundation for such a facility.

The interpreter modules Recognize and Assume are used in a dialogue environment. Assume handles method steps that are carried out by the user but which have no corresponding input to the console, e.g., reasoning steps. Recognize is responsible for calling the parser, then matching parser output against the expectations that it develops and maintains. Both Recognize and Assume depend heavily on special OWL conversational methods which will be described in the next section.

Two other major modules are the parser and the generator. The dialogue routines are not currently interfaced to these modules, so I will not discuss them in detail in this paper. I will take this opportunity, however, to make a distinction between *interpreter level* and *surface semantic* OWL representations. As the name implies, interpreter level representation is used by the interpreter and is the stuff of which methods are made. Surface semantic representation is output by the parser and is also input by the generator. The difference between the two is that interpreter level representation has undergone more canonicalization than its surface semantic counterpart. In general, where a surface semantic representation will look very much like its surface English counterpart, an interpreter level version of the same utterance will have referents substituted for referring expressions and will have undergone more lexical standardization. While I will not go into further detail here, the distinction is sufficient to make the point that the output of the parser and the input to the generator are not at the same level of representation as that used by the interpreter. Two intermediate modules are necessary to provide the translation between them: the reference finder and the pre-generator. The reference finder takes the surface semantic representation output by the parser and looks for corresponding interpreter level referents, both substantives and events. The pre-generator goes in the other direction, taking interpreter level concepts and finding descriptions for them and ways to express them so that the user will be able to identify the sense intended.

This accounts for the major OWL modules, and we turn now to a short overview of notation. As mentioned above, there are two fundamental operations in OWL, specialization and modification. Specializations of a concept are represented using parentheses, e.g., (NAME FIRST) for "first name," a specialization of NAME. FIRST is called the *specializer* of (NAME FIRST). To represent the fact that a concept modifies another concept, we use square brackets to form a *complex*, e.g., [PAPER OFFICE-SUPPLY]. This says that the concept for paper has the concept for office supply as a modification. Note that OFFICE-SUPPLY is actually a *label* for a concept that might also, for example, be written as (SUPPLY OFFICE). In general, labels are used to increase readability, and they are assigned using an equal sign, e.g., OFFICE-SUPPLY=(SUPPLY OFFICE). A special position on the reference list is reserved for *values* of relational concepts such as EMPLOYER, SUPPLIER, LENGTH, WIDTH, etc. The notation for value assignment is a left arrow; for example,

```
(EMPLOYER MARY-DOE) <-- UNION-CARBIDE
```

says that the employer of Mary Doe is Union Carbide. Mechanisms exist in the interpreter to handle values that change over time and also to handle values that are context or world model dependent.

Colons are used in OWL as an abbreviation for specialization by the first concept (the *subject*) of a complex. Thus,

```
[BLOCK-A   COLOR: <-- RED]
```

is equivalent to:

```
[BLOCK-A   (COLOR BLOCK-A) <-- RED]
```

Both say that the color of Block-A is red. The number of colons corresponds to the level of embedding of the square brackets, so that on input the expression

```
[((PUT ENTITY) ((ON TOP) ENTITY))
  INSTRUMENT: <-- [HAND:
                    (PART AGENT:) <-- ::]]
```

would be equivalent to

```
[((PUT ENTITY) ((ON TOP) ENTITY))
 (INSTRUMENT ((PUT ENTITY) ((ON TOP) ENTITY))) <--
     [(HAND ((PUT ENTITY) ((ON TOP) ENTITY)))
       (PART (AGENT ((PUT ENTITY) ((ON TOP) ENTITY)))) <--
          (HAND ((PUT ENTITY) ((ON TOP) ENTITY)))]]
```

Both OWL structures above express the constraint that the instrument case of the concept ((PUT ENTITY) ((ON TOP) ENTITY)) must be bound to the agent's hand. Specialization by the subject of a complex is used to tie concepts into larger OWL structures. For the OWL interpreter, colons most often indicate that a concept is to be used as a variable.

We are now in a position to look at figure 3 in more detail. The subject of the whole complex is ((PUT ENTITY) ((ON TOP) ENTITY)); this means that the other concepts that follow will appear on its reference list. I have called this particular subject the header. From the point of view of the OWL system base, all of the concepts on the reference list of the PUT header are about the same. For the interpreter, however, there are important semantic distinctions between them, some of which have already been mentioned. The first of these concepts is PLAN. This is a category used by the OWL interpreter for type-checking purposes; it merely distinguishes some generic concepts from instances, in this case PUT events. (In this discussion, I will refer to the PUT concept, although technically the concept we are dealing with is ((PUT ENTITY)((ON TOP) ENTITY)).) Next come the semantic case specifications; for activities, these come from a set of 20 cases (which may, however, be further specialized). Semantic case names are used as variables elsewhere in the method, so that INSTRUMENT:, for example, refers to the concept bound to the variable that is the value of the instrument case, in this example HAND. The case specifications of figure 3 say that in order for the PUT activity to run, the OBJECT must be bound to a kind of ENTITY (the thing to be moved). Here, the ":1" is used to distinguish this entity variable from others. The AGENT must be bound to a kind of person or computer system, which in this case will be the computer system. The INSTRUMENT must be bound to the AGENT's hand and the SPECIFIC-LOCATION will be on top of another entity. The PRINCIPAL-RESULT, an output case specification, indicates that the location of the OBJECT (that is, the concept bound to ENTITY:1) will be at the SPECIFIC-LOCATION (that is, on top of another entity). Note the use of semantic case variables in the statement of this PRINCIPAL-RESULT.

The semantic case specifications are followed by the steps in the method, separated by commas. (Other notation -- the use of THEN concepts -- may be used when the linear ordering supplied by commas is insufficient.) These steps are calls that can be carried out using either LISP procedures or other OWL methods. The semi-colons followed by numbers here are comments. An English version of the PUT method goes as follows: first, display the scene (a graphics display of the blocks world is operational), then find a space for the OBJECT at the SPECIFIC-LOCATION. The SOME in the second step indicates that the variable SPACE: is *nonspecific*, that is, it will not be bound until the FIND method has been carried out. After a space is found, the OBJECT is grasped and the scene is redisplayed. THE specializers, as in step 3, are used by the interpreter to constrain some of the knowledge base retrievals. Due to the highly interwoven nature of the knowledge base, there are some situations in which a search for constraints on a concept turns up not only constraints, but also a set of related concepts. I will not go into the mechanism in detail, but note that by using THE, we create a new concept, assuring that only constraints are retrieved when they are desired.

The fifth step in the PUT method is to MOVE the HAND to a place located on the X-Y grid above the SPACE found in step 2. At this point, the PRINCIPAL-RESULT can be asserted, the scene is redisplayed, the AGENT lets go of the OBJECT, and a final display is done. Note that the steps in the PUT method lack some of their case assignments. These can be inherited from higher events, so that the AGENT of each of the nine steps is assumed by the interpreter to be the concept bound to the AGENT of ((PUT ENTITY) ((ON TOP) ENTITY)). Finally, to avoid confusion I should mention the numbers distinguishing the different DISPLAY calls. These are necessary because a plain (DISPLAY SCENE) would not be tied into the PUT method, and adding a colon to do this would still make temporal order unrecoverable in the internal representation. We therefore use :1, :2, etc. Currently, this is the only exception to the rule that colons mean variables to the interpreter.

From this example it should be clear that OWL representations have different significance at different layers of the system, and that OWL notation, correspondingly, has more than one layer. The OWL system base deals in concepts, specializations, and complexes which the user represents with parentheses, colons, square brackets, etc. The interpreter, on the other hand, has a higher level, more semantic, point of view, and it deals with semantic cases, procedure calls, variables, and so forth. Finally, there is the surface semantic level of representation which is the OWL used by the English parser and generator.

This completes the overview of the OWL system, and we can now turn to issues of processing dialogue.

## 3. Conversational Exchanges

### 3.1. What They are and How to Model Them

Looking at the sample dialogue, we can pick out groups of lines that seem to belong together. An example of such a unit would be a question plus its answer:

What is your name?
◊ Bill

This is an example of what I will call a *conversational exchange*. At the more complex end of the conversational exchange scale (although outside the scope of the Susie system), would be an exchange like formal debate. A debate program might consist of several subsections and many steps. I see the conversational exchange as a basic structure at the interpersonal level of dialogue. This places them at a higher level than speech acts <Searle>, which do exist in a conversational environment but which are carried out by a single agent. For example, promising and accepting the promise would be two speech acts, but they would form a single conversational exchange.

This is not to say, of course, that dialogues can only proceed by relentlessly completing conversational exchanges. Many real-life dialogues are much less orderly, their structure influenced by a set of competing goals. In <Carbonell> we see an example of a conversational exchange (a question-answer sequence) that remains temporarily uncompleted:

Approx what is the area of Argentina?

◊Tell me something about Peru◊

Peru is a country.
It is located in South America.
The capital of Peru is Lima.

Now answer the question you didn't answer before.

◊The area of Argentina is 1,100,000 square miles.◊

Thus, while I see the notion of a conversational exchange as an important one, I by no means advocate a rigid application of it. We will see that the Susie system uses the expectations set up by a conversational exchange in attempting to understand a user input, but these expectations are used in a very flexible way.

In this section I will describe the way that conversational exchanges are modelled in the Susie system, and then in the next section I will develop some other distinctions necessary to actually use these structures to model dialogue.

Conversational exchanges are modelled using OWL methods. Since the question-answer exchange is a relatively simple example, I will pursue it further. Figure 4.1 gives the English version of the procedure ASK-AND-ANSWER and figure 4.2 shows a simplified version of the OWL method.

```
ask-and-answer

        object: the question to be asked
        agent: a person or computer system
        co-agent: a person or computer system

        method:

         1. The agent asks the question.
         2. The co-agent now knows what the question is.
         3. The co-agent finds the answer.
         4. The co-agent gives the answer and the agent gives an
            (optional) acknowledgement.
```

Figure 4.1

```
[ASK-AND-ANSWER
 PLAN
 OBJECT: <-- [SUMMUM-GENUS:1 NON-HOW-WHY-QUESTION
          ((BE ((INFORMATIONALLY-NONSPECIFIC -SELF)
                      (FOR CO-AGENT:))) ::)]
 AGENT:    <-- (OR HUMAN:1 VERBALIZER:1)
 CO-AGENT: <-- (OR HUMAN:2 VERBALIZER:2)

METHOD: <--
;;;1
 [(ASK OBJECT:)
      AGENT::        <--  AGENT:
      DESTINATION::  <--  CO-AGENT:],

;;;2
 (BECOME
     ((BE (SPECIFIC -SELF)) '(OBJECT: (FOR CO-AGENT:)))),

;;;3
 [((FIND MENTAL)
         ([SUMMUM-GENUS:3  (ANSWER OBJECT:) <-- :::]
          SOME))
      AGENT::        <--  CO-AGENT:
      BENEFICIARY::  <--  AGENT:],

;;;4
  [(STATE-AND-ACKNOWLEDGE  (SUMMUM-GENUS:3 THE))
      AGENT::     <--  CO-AGENT:
      CO-AGENT::  <--  AGENT:]]
```

Figure 4.2

The basic form of this procedure is very much like that of the PUT method discussed in the last section. The header, ASK-AND-ANSWER, is followed by specifications for type checking of the input cases, and then the rest of the OWL representation is devoted to a set of procedure steps and their case assignments. By convention, the AGENT of the entire conversational method is the AGENT of the first step, so that we can identify the AGENT of the method with its initiator. Looking at the OBJECT specification, we see that ASK-AND-ANSWER handles all questions besides how- and why-questions (the NON-HOW-WHY-QUESTION constraint). These other two varieties of question are handled by ASK-AND-DESCRIBE and ASK-AND-EXPLAIN, two methods that will be discussed further later on in this section. The primary class of the OBJECT is SUMMUM-GENUS, the top node on the concept tree; it is used here because NON-HOW-WHY-QUESTION is a secondary, rather than a primary, characterization. The other rather involved constraint on SUMMUM-GENUS:1 says that for the CO-AGENT the OBJECT is *informationally nonspecific*. This term means that on entry to the ASK-AND-ANSWER method, the CO-AGENT cannot bind the OBJECT to the question because he does not know it yet. (A variable may also be *existentially nonspecific*, i.e., it cannot be bound because the intended binding does not exist yet.) The -SELF specializer is present because specificity is treated as a property of variables, rather than their bindings; that is, (INFORMATIONALLY-NONSPECIFIC -SELF) is a property of SUMMUM-GENUS:1, not of a prospective binding. Note that in step 2 of the method, after the question is asked, the CO-AGENT is assumed to know the question, and an assertion is made about the new state of his knowledge base.

An important point about the ASK-AND-ANSWER example is that it contains the parts played by both speakers and is intended for use by the interpreter whether it is Susie or the user who is the one to initiate the exchange by asking the question. I will call this latter property *speaker independence*. If the conversational methods are speaker independent, then it is up to the OWL interpreter to determine whether a particular answer is to be generated or understood. There are, in fact, three possible modes of interpretation[3] for a step in a conversational method:

(1) Carry out the step (e.g., ask a question).

(2) Recognize that a step has happened (e.g., that an answer to your question has been given).

(3) Assume that a step has happened (e.g., if your conversational partner gave the answer, then he had to perform the mental process of finding the answer first.)

Given the input case settings in a call, the interpreter uses a set of simple rules to determine the mode of a step.

3. A difference of terminology here. Throughout this paper, the words *interpret* and *interpretation* will be used to indicate actions of the OWL interpreter. The meaning of *interpret* found in *natural language interpretation* (as opposed to *natural language generation*) will be rendered by *recognize* and its variants.

One final general point about the conversational methods is their abbreviated form. For any given speech act, only production is represented explicitly, and the activities of the other partner are left implicit. Thus, although the Susie programs are speaker independent in the technical sense defined, they are not without a bias: it does not matter who is specified as the agent of a communication step, but, whoever is, the "story" is told from his point of view.[4] "Listening" steps are left implicit not because they are unimportant, but because the form and timing are predictable. Where a joint model of communication is necessary (e.g., when misunderstandings occur) the interpreter can expand the abbreviated model expressed in the conversational methods.

In the terms of current artificial intelligence research, the conversational methods amount to frames for conversational exchanges. The highly centralized control structure of the OWL interpreter, however, differentiates it from many existing frame systems. The OWL conversational methods themselves are probably most similar to the social action paradigms in <Bruce>; the primary difference is the distinction that will be developed in the next section between standard and failure paths.

## 3.2. Analyzing the Sample Dialogue

In addition to the ASK-AND-ANSWER method, the Susie system currently contains thirty other conversational methods needed either to run the sample dialogue directly or to provide a rich enough environment to test the procedure selection and matching routines. (Any of the methods that have been written to carry out dialogue will be referred to here as conversational methods, even though some of these may have applications in non-conversational tasks as well.) There is not space here to discuss the conversational methods in detail, but to give an idea of their content I will describe the way that they are used to model the sample dialogue of figure 1. Implicit in this description will be my analysis of the conversational exchange structure of the dialogue.

The sample dialogue is first of all a console session, so we have the method ((PARTICIPATE IN)(SESSION CONSOLE)). The first steps of this procedure handle the greeting and introductions (lines 1-2) and the last step handles the closing (line 26). In the middle of the PARTICIPATE method is the call to carry out one of Susie's two top level activities, writing a program or answering a question. This core step is repeatable an indefinite number of times. The sample dialogue shows both of the top level activities: lines 3-23 contain a program writing exchange, and lines 24-25 contain a question answering exchange.

4. In describing the "active" bias of the method representation style, I do not mean to imply that the rest of the system shares this bias. The event tree maintained by the interpreter, does not share the bias; moreover, neither does the interpreter itself.

Since the question-answer exchange is the simpler, I will discuss it first. The three main ways to get a question answered in the Susie system are ASK-AND-ANSWER, ASK-AND-DESCRIBE, and ASK-AND-EXPLAIN. As mentioned above, ASK-AND-ANSWER handles most what-, where-, whether-, and when-questions. Why-questions are handled by ASK-AND-EXPLAIN, and how-questions are split between ASK-AND-DESCRIBE and ASK-AND-EXPLAIN depending on the type of information that seems appropriate. Of course ASK-AND-DESCRIBE and ASK-AND-EXPLAIN can also be triggered by a direct request for a description or explanation, respectively. The motivation for distinguishing ASK-AND-DESCRIBE and ASK-AND-EXPLAIN from ASK-AND-ANSWER is that the first two will tend to be involved with longer answers that require more selection and organization of the information. ASK-AND-DESCRIBE and ASK-AND-EXPLAIN are distinguished from each other by the aspects of the topic that are considered relevant; for ASK-AND-EXPLAIN, the emphasis is on causal relationships.

In the sample dialogue, the user question "How did you decide to use your left hand to move B2?" is on the borderline between ASK-AND-DESCRIBE and ASK-AND-EXPLAIN. To see this, compare:

(i) Explain how you decided to use your left hand to move B2.
(ii) Describe how you decided to use your left hand to move B2.

A description of the decision process would list the temporal sequence of events, while an explanation would list the causal sequence. Since in this case the two sequences are identical, it does not particularly matter how the answer is generated. In the Susie system, I have assigned this type of how-question to ASK-AND-DESCRIBE. ASK-AND-DESCRIBE gathers and orders information and then calls STATE-AND-ACKNOWLEDGE. In this procedure, the AGENT makes a statement and the CO-AGENT, optionally, acknowledges it.

We turn now to the program writing exchange, which starts at line 3 with the user asking for a program. This line triggers COMMAND-AND-RESPOND, a method that handles nonverbal activities done by one person (or system) for another. (It also handles the case of a verbal activity where the recipient of the information is not the one who asked for the activity, e.g., "Tell Harry what you told me.") COMMAND-AND-RESPOND is designed for any authority relationship between participants, so that it also handles requests. In lines 4-5 Susie finds that her capabilities are not as broad as the user's general request, and an attempt is made to get a more specific idea of what the user wants. These lines are treated as a temporary departure from the COMMAND-AND-RESPOND method onto what will be called a *recovery path*. The notion of a recovery path and the motivation for it will be discussed in the next section.

Once the user's request is clarified, the system enters the (WRITE PROGRAM) method. In this procedure, conversational steps are intermixed with non-conversational ones, i.e., the actual program-writing calls. In the Susie system there is no sharp distinction between the representation of methods designed solely for dialogue, methods that use dialogue to gather information in order to get other work done, and non-dialogue methods such as block manipulation routines. Susie's first step in writing a program is a call to (GET DESCRIPTION), where there are currently two alternatives. If the user probably has no idea of the properties of the input and output that are of interest, he or she can be guided through the description by a series of ASK-AND-ANSWERs for which Susie generates questions. If the user is assumed to know the relevant aspects, as is the case in the sample dialogue, then a subcall is made to ASK-AND-DESCRIBE. In the sample, the request and the subsequent description constitute the exchanges from lines 6 to 17.

With input and output conditions described, Susie can now go on to write the program. More information is needed, however, so she returns to the user with a question-answer exchange handled by ASK-AND-ANSWER, lines 18-20. When the program is finished, Susie notifies the user and then does an ASK-AND-ANSWER to find out whether a description of it is wanted (lines 21-22). Since in this case the user does want a description, that becomes the final step of the (WRITE PROGRAM) method (line 23).

We have seen how the idea of a conversational exchange and the corresponding conversational methods can be used to analyze the structure of a dialogue. This is only part of the story, however, as the allusion to recovery paths indicates. More distinctions and additional mechanisms are necessary, and these will be introduced in the next section.

## 4. Basic Utterance Types

Since we cannot expect all dialogues to consist of a chain of well-formed conversational exchanges, one following another, a system must be able to handle dialogues that may differ from this model. As a step in this direction, I will distinguish four basic classes of utterance.

The first utterance type is the one that has already been considered: an utterance may correspond to a step in a task that is already underway. I will call this a *standard path successor step*. An example would be the answer to a question or the acknowledgement of a statement.

Next, in a task environment the tasks must start off in some way, and I will call the class of utterances that may initiate a task *top level lead-ins*. Recall that in the Susie Software environment there are two types of tasks, program writing and question answering. (Each of these may of course be composed of subtasks.) In this environment, then, top level lead-ins are either requests for a program or for information. Note that not all conversational exchanges are initiated by top level lead-ins. An utterance corresponding to the first step in STATE-AND-ACKNOWLEDGE, for example, might be called a lead-in, but it would not be a top level lead-in, since it would not initiate one of Susie's two top level activities.

A third basic utterance type is *failure discussion*. The standard path of a conversational method is intended to specify the relatively small number of ways that an exchange can be concluded successfully. This is fine as long as the dialogue goes as intended and no expectations are violated. In practice this will probably not be long, and failure discussion will result.

Let us look at an example from the sample dialogue:

6 O.K. I need a description of the initial and final scenes.
◊ The final scene contains 3 blocks: Bl, B2, and B3.
O.K.
◊ What else do you want to know about the final scene?
10 What are the positions of Bl, B2, and B3 with respect to each other?
◊ Bl is on top of B2 and B3 is on top of Bl.
O.K.
◊ Initially, all of them are on the table.

As I analyze this section, in line 6 Susie asks for a description and in line 7 the user starts to give it. This is implemented with the method ASK-AND-DESCRIBE. At line 9 the user indicates that his model of what Susie wants to know is insufficient. When this happens, Susie shifts to another strategy which accomplishes the same goal but which assumes less knowledge on the part of the user. This is reflected in line 10, where Susie asks a question, thereby communicating what it is she wants to know. By line 12, the difficulty has been cleared up, and the dialogue is back on the standard path of the ask-and-describe exchange.

Among the failure conditions that will generate discussion, I have concentrated on lack of the information necessary to make a decision, since this is the case that comes up in the Susie dialogue. Other failure discussion may come up as a result of contradictions (in a sense, the overabundance of information) and misunderstandings. To model some of the failure discussion that will come up in the course of dialogue, I have introduced a structure called a *recovery path*.

Examples of recovery paths are shown in figure 5.2, which is a more complete version of the ASK-AND-ANSWER method introduced in section 3. It is preceded by figure 5.1, its English translation.

ask-and-answer

object: the question to be asked
agent: a person or computer system
co-agent: a person or computer system

method:

1. The agent asks the question.
2. The co-agent now knows what the question is.
3. The co-agent finds the answer.
4. The co-agent gives the answer and the agent gives an (optional) acknowledgement.

recovery path 1: if a stipulation is found along with the answer
R1.1 The co-agent states the stipulation.
R1.2 The agent agrees to it.

recovery path 2: if the answer is unknown
R2.1 The co-agent says that he doesn't know the answer.

Figure 5.1

```
[ASK-AND-ANSWER
 PLAN
 OBJECT: <-- [SUMMUM-GENUS:1  NON-HOW-WHY-QUESTION
          ((BE ((INFORMATIONALLY-NONSPECIFIC -SELF)
                            (FOR CO-AGENT:))) ::)]
 AGENT:    <--  (OR HUMAN:1 VERBALIZER:1)
 CO-AGENT: <--  (OR HUMAN:2 VERBALIZER:2)

 METHOD: <--
  [(ASK OBJECT:)
       AGENT::       <--  AGENT:
       DESTINATION:: <--  CO-AGENT:],

  (BECOME
      ((BE (SPECIFIC -SELF)) '(OBJECT: (FOR CO-AGENT:)))),

  [((FIND MENTAL)
          ([SUMMUM-GENUS:3   (ANSWER OBJECT:) <--  :::]
            SOME))
       AGENT::       <--  CO-AGENT:
       BENEFICIARY:: <--  AGENT:],

   [(STATE-AND-ACKNOWLEDGE  (SUMMUM-GENUS:3 THE))
        AGENT::    <--  CO-AGENT:
        CO-AGENT:: <--  AGENT:]

 ((RECOVERY-PATH STIPULATION) ((FIND MENTAL) SUMMUM-GENUS:3)):
      <--
      [(TELL (AND (SUMMUM-GENUS:3 THE)
               (STIPULATION (PRINCIPAL-RESULT
                    ((FIND MENTAL) (SUMMUM-GENUS:3 THE)))?)?))
        AGENT::       <--  CO-AGENT:
        DESTINATION:: <--   AGENT:],

      [(TELL [SUMMUM-GENUS:4 AFFIRMATIVE])
        AGENT::       <--  CO-AGENT:
        DESTINATION:: <--  AGENT:]

 ((RECOVERY-PATH (KNOW NOT)) ((FIND MENTAL) SUMMUM-GENUS:3)):
      <--
      [(TELL (AND (BE SORRY):1
               ((KNOW NOT) (WHAT (ANSWER OBJECT:)))))
         AGENT::       <--  CO-AGENT:
         DESTINATION:: <--  AGENT:]]
```

Figure 5.2

Here, ASK-AND-ANSWER has two recovery paths. The first, containing the concept STIPULATION, handles the situation where an answer can only be given if there is an associated stipulation, e.g. line 19 of the sample dialogue, "Yes *if I can ask you about them later?*" The second recovery path handles the case where no answer can be found. Both of these failures occur in the process of finding the answer, step 3, but the recovery path is associated with ASK-AND-ANSWER. A different recovery path would be used if the find answer routine were called in another context, e.g., reasoning.

Recovery paths are a very local way to model failure discussion, and they are not expected to be useful for all cases where expectations are violated. For example, an ambiguity may trigger failure discussion (of the "What do you mean ..." variety), and ambiguities can be generated in the process of any sort of English output. This type of failure discussion, then, would be better handled by an autonomous OWL method, rather than a recovery path. Such autonomous methods are part of the general OWL failure mechanism. Note that the sample dialogue does not contain any lines that I would model with the general failure mechanism, so that this possibility will not be considered in detail in this paper.

Turning from failure discussion, the fourth basic utterance type is *metadiscussion*. Utterances classified as metadiscussion deal with the conversational situation itself. Based on the dialogues I have looked at, it seems to me that true metadiscussion turns out to involve a relatively narrow range of utterances. Many utterances that I would initially class as metadiscussion because they deal with the conditions of conversation turn out, on closer examination, to be better classified as failure discussion. For the Susie Software environment, I have found only three categories of utterances that are purely metadiscussion. The user can either suspend an activity ("Let's stop this for now."), reopen a suspended or closed one ("I want to go back to the first program you wrote."), or specify what he or she is going to do next ("Now I'll tell you what I'm going to do"). Note that as I analyze it, the sample dialogue does not contain any examples of metadiscussion, although metadiscussion has been handled in the design of the recognition process.

This finishes the discussion of the four basic utterance types: standard path successor step, top level lead-in, failure discussion, and metadiscussion. The distinction will be important to the system when it comes time to develop structural expectations about the form of a user input, and this distinction will form the basis for a mixed approach to matching for recognition.

## 5. The Interpreter Modes Carry-out and Assume

In section 3 a distinction was made between three modes of method interpretation: carry-out, assumption, and recognition. In this paper I am focussing on the recognition process, but before looking at it in detail I will make a few observations about the other two modes of interpretation.

Carry-out mode has already been described briefly in section 2. Its main components are the method selection apparatus, the event tree maintenance routines, and the set of failure mechanisms. The Carry-out module is not unique to dialogue, although dialogue does exploit some of its most important capabilities. The property of carry-out mode that is most important for dialogue is the fact that information on the event tree is never lost, so that a complete control record is available. A more thorough discussion of Carry-out may be found in <Sunguroff>.

The second interpreter mode, assumption mode, was introduced to handle dialogue, but it is only rudimentary at this time. There are a number of complex and interesting issues that surround assumptions about the knowledge and mental processes of others (e.g., issues of level of detail). We felt, however, that we lacked information upon which decisions about assumption mode could be based. We have therefore deferred work in this area until we have more experience with the reasoning processes that will use the assumptions. This experience should begin to clarify level of detail issues. For the present, the interpreter notes an assumption mode call on the event tree by putting it on the subevent list of the event of which it is a subcall. There is no search for a matching method or creation of an event for the assumption call. This minimal treatment allows the interpreter to hold onto the information that a particular substep has been assumed, giving it an unbroken record of the paths taken through the methods that have been executed.

This is a very brief treatment of carry-out and assumption mode, but it is sufficient for our purposes here. Having looked at conversational exchanges and some basic distinctions among utterance types, we are now ready to consider recognition mode.

## 6. An Outline of Recognition Strategy

Given an English language input, Susie Software must find or construct a corresponding interpreter level representation, since this level of representation is the one used in the system's reasoning processes. Within our framework, then, the transition from English to interpreter level OWL constitutes the recognition process. In this section I will outline the strategy used for recognition, starting with a closer examination of the problem.

The basic problem for recognition is the overabundance of alternatives. If the process is broken down into subprocesses, each subprocess may produce several different possible interpretations of the input. At any phase of processing, the environment may be rich enough to present a number of alternatives. When multiple results from one phase meet multiple alternatives in the next one, we have the potential for combinatoric explosion.

In the Susie Software system recognition is divided into three subprocesses: parsing, matching, and expectation management. Each has the potential to produce multiple alternatives. An input may have more than one parse, and the results of a parse may contain referring expressions that match more than one referent. Moreover, in the Susie environment there are several degrees of flexibility that enrich the set of possible structural expectations. Decisions about what-happens-when are not completely controlled by the system. Susie Software offers a mixed-initiative environment, allowing the user a say in the timing of choices. This moves the Susie environment in the direction of normal conversation, since, if either participant may change the flow of control, the other participant will have less than complete knowledge about what will happen at any given point. In addition, activities in the Susie environment are not rigidly ordered and disposed of. A new activity may be begun before the old one has been completed, and an activity may be reopened after it has been assumed to be finished. Furthermore, the kinds of exchanges that may occur make it harder to find the boundaries between activities. Giving a description for example, is open-ended in a way that a multiple choice answer would not be.

Finally, as we have seen, discussion may occur on more than one level. We not only have utterances that relate to the ongoing task directly, but also utterances that report failure conditions in the ongoing task and metadiscussion, that is, utterances that explicitly alter or clarify the flow of activity. Moreover, the user may initiate a totally new task at any time.

These combinatoric problems have been foremost in our minds as we have explored different approaches to recognition. The parsing strategy developed by Szolovits and Martin intentionally limits the extent of processing, at least by comparison to existing systems (e.g., <Woods> and <Winograd>). As the extent of processing is reduced, so are the accompanying combinatorics. For the matching of parser output to interpreter level representations, we are attempting to constrain the process first by a set of heuristics about the position of likely referents. In addition, we hope to keep matching problems under control by the adoption of a mixed matching strategy: different matching schemes have been chosen for the four utterance types distinguished in section 4, and there are also distinctions made within these types.

One very crucial factor in the attempt to reduce combinatorics is the make-up of the surface semantic representation output by the parser. This level of representation is examined in <Martin>; basically, the concepts in surface representations are chosen to minimize the number of decisions that must be made by the parser. In particular, the parser does not attempt to make distinctions that are not needed to complete the parse. This philosophy is similar to the approach taken in <Marcus>, but it goes beyond it in the extent to which decisions are delayed.

To illustrate the decision-delaying nature of the surface semantic representation, we can consider the different ways to say that one understands some information. One informal way is to say, "I get it." Now, "I get it" in isolation is ambiguous (e.g. Q. "Do you know anyone who gets this journal?" A. "I get it."). A transformation within the parser would have to expand GET into its alternatives, say RECEIVE and (GET IDEA), which is not in the spirit of a decision-delaying surface representation. It therefore seems best to use GET in the surface semantic representation, then depend on interpreter level semantic structures to make further distinctions. Appropriate structures will be suggested in the sections that follow.

Since the parser implementation is still in progress, I will not discuss it further here. The reference finder implementation is also incomplete at this time, but a few words about it are necessary to put the structural expectation matching processes into perspective. Recall that the surface semantic representation output by the parser contains OWL representations corresponding to referring phrases, and references have not yet been resolved. The reference finding process starts with this representation. The basic philosophy for reference finding has been to exploit both the structure built up on the event tree and the structural expectations (especially the current set of possible standard path successor steps). A natural distinction arises between references to interpreter level concepts that are either explicitly given or already instantiated in the structural expectations and references to interpreter level concepts that are not part of these expectations. The latter may either be present earlier in the dialogue (and thus available on the event tree), part of general knowledge and independent of the current dialogue, or they may be found in the course of processing the utterance (i.e., forward references).

The implementation currently handles referents present in the structural expectations, but it does not yet handle more global references. (It appears that the sample dialogue could be handled in such a way that all references can be treated as references to concepts in the structural expectations.) Work is continuing on the reference finder, but the most important characteristic is already evident: for both types of reference finding, the process is integrated into the process of matching against structural expectations. (Whether particular referents are present explicitly or not, the interpreter will always be matching a user input against *some* structural expectation.) Thus, while parsing happens in an identifiably separate pass, reference finding occurs as needed within the general process of matching surface semantic representations against interpreter level forms.

Having outlined the parsing and reference finding strategies, I can now discuss the way that structural expectations are managed by the system. Recall that distinctions among basic utterance types were made in section 3. From special patterns that will be described and from the conversational methods, the system can derive a set of structural expectations at any given point in the dialogue. The question is, what should the system do with these expectations?

The first issue for expectation matching is the choice between a try-all-possibilities and a stop-on-success strategy. Our original strategy for handling the different possible types of next steps in recognition mode was to try all possibilities. Based on my experience with the sample dialogue, however, I have concluded that this would involve a great deal of time and that, if the sample is representative, a stop-on-success scheme will be sufficient as long as we are careful in the way that attempts are ordered. The change in strategy is worth discussing, I think, because it points up some important aspects of the recognition process.

First, in a try-all-possibilities environment, the burden is placed on disambiguation, while in a stop-on-success environment it is placed on ordering. When trying all possibilities, if matching leaves ambiguities then there are two main sources of information: heuristics and the conversational partner. The goal of a stop-on-success scheme, then, should be to incorporate these information sources. If this can be done (at least a large part of the time), then the stop-on-success scheme can perform as well as trying all possibilities and save time in the process.

Looking first at information from the conversational partner, for a disambiguation process the standard mechanism would be to ask for clarification. A stop-on-success scheme, on the other hand, would depend on the partner's ability to catch incorrect interpretations from the responses given. For our constrained environment it looks like we will have no difficulty in framing responses in such a way that the user will know whether or not his intentions were interpreted correctly. If a misinterpretation does occur, the user's next utterance will be something on the order of "That's not what I meant." A general failure method, or a small set of them, can be used to handle this situation.

The second information source is heuristics, and I will give two examples here. The first is a redundancy heuristic: a speaker should not (and therefore usually will not) perform a speech act which is embedded in an essentially identical speech act. Thus, where an input line fits both a standard or recovery path in an open method and also could be construed as initiating a new task, then the standard or recovery path should be considered to be the correct match. This redundancy heuristic is very much like one for objects in a description: if there is no information to the contrary, similar identifying phrases can usually be assumed to have identical referents.

A second heuristic can be called the "inertia" heuristic: all else being equal, a context will tend to persist. Given any ambiguity between top level lead-ins and standard or recovery path steps, I think the interpreter should assume that the latter is intended. This course of action is obviously not going to be infallible, but I think that we will see it successful more often than not.

Now, in a try-all-possibilities scheme, heuristics of this sort would either be built into the interpreter or into special disambiguation routines. On the other hand, in a stop-on-success scheme the heuristics would be reflected in the ordering rules chosen. For example, both heuristics given can be incorporated into an ordering scheme easily enough by requiring that standard and recovery path expectations be checked before top level lead-ins. Since ordering is crucial to a stop-on-success approach, I will finish up this overview of recognition strategy with a discussion of the ordering used in the system.

Based on the distinctions made in section 3, an input from the user can be one of several different types: top level lead-in, standard path successor step, metadiscussion, lead-in to a recovery path, successor step on a recovery path, lead-in to a general failure method, or successor step on a general failure method. The following list sums up the ordering of recognition possibilities used in the current implementation if no failure discussion is underway:

(1) Metadiscussion

(2) Standard path successor steps

(3) Recovery path lead-ins

(4) Top level lead-ins

(5) General failure method lead-ins

Once a recovery path or general failure lead-in has been processed (either recognized or generated) then the relevant successor steps become expectations and are checked second, in place of standard path successor steps. Note that this ordering scheme has worked well so far, but it is intended as a hypothesis only; the implementation is structured so that different ordering arrangements can be tried easily.

To finish out this section, I will make a few observations to motivate the ordering scheme given above. The first utterance type to be checked for is metadiscussion. The range of the class seems to be constrained enough that only a few patterns need be matched, and, since the nature of metadiscussion is to alter or clarify the flow of control, it seems reasonable to check for this possibility before anything is done to continue in the normal flow of control.

Next come standard path successor steps and recovery lead-ins. These are the expectations that vary most as the dialogue progresses; once a conversational method is underway, it is likely that the user will continue in that track, so it seems reasonable to check these early. Since there are many more ways that things can go wrong than right, I have put the standard path successor steps first. We may, however, want to try interleaving recovery path lead-ins with the standard path successors, so that a standard path successor would be checked and then any recovery paths related to it before the next standard path successor was tried.

The next basic utterance type is the top level lead-in. The heuristics given above indicate that this utterance class should be tried after standard path successor steps. In terms of efficiency, also, it makes sense to put top level lead-ins near the end. As section 7 will show, even for as constrained an environment as the Susie Software one there are a number of possibilities in this class. Because the top level lead-in possibilities will not be constrained by the progress of the dialogue, an unsuccessful top level lead-in match will be a relatively expensive operation. Checking for top level lead-ins early on, then, would probably be inefficient on the whole.

Last come the lead-ins to general failure methods. Since none of these are present in the sample dialogue, I have the least experience with this category. It may be that we will want to try these before top level lead-ins. The final decision will be based on experience with matching strategies for the two classes. Basically, whichever match attempt fails fastest on the average (as long as worst cases are not unreasonable) should be tried first.

This winds up the discussion of ordering. As we have seen the basic recognition strategy has been chosen with combinatoric problems in mind. The parser delays as many decisions as possible, the reference finder takes advantage of expectations developed from the conversational methods, and match attempts are carefully ordered. Another very important element of the recognition strategy is the mixed matching scheme that has been developed. This scheme is based on the distinctions between basic utterance types that have been presented, and it is the topic of the rest of this paper.

A final note on the discussion that follows: I will make the simplifying assumption that only one surface representation has been produced for an input. Where more than one surface representation is output by the parser, attempts are made to match each of them against the relevant structural expectations. At some point, the interpreter may become clever enough to know when to stop looking for additional matches, but for now if the combinatorics do not become prohibitive (and I doubt that they will, given the limitations on the Susie environment) all parser outputs will be tried.

## 7. Recognizing When a New Task is Initiated

If Susie is the AGENT of the first step in a method, then she knows both the goal she is trying to achieve and the events superior to the step on the event tree. If, on the other hand, the user is the AGENT of this first step, then it may be the case that both goal and superior events are not derivable from the preceding context. This would happen if the step were the lead-in of an activity to achieve one of Susie's two top level goals: providing the user with a program or with an answer to a question. Such initial utterances, and the conversational method steps that generate them, have been called top level lead-ins. (Note that a lead-in step may not necessarily correspond to the initial step in a conversational method but is instead the first step to produce an input for Susie; the initial step may be one that is executed by Susie in assumption mode.)

To process a top level lead-in in recognition mode the interpreter needs the ability to form a chain of event links between the top node on the event tree and the event that corresponds to the user's utterance. For example, assume that the only currently open task is to have a console session and the parser gets the input "Can you pick up the block?". The surface semantic representation output by the parser might be:

[YOU / (CAN ((PICK UP) (THE BLOCK))) QUESTION]

The task of the interpreter would be to find a generation step that could have produced this utterance, as well as a chain of containing events that connect the current environment -- the (PARTICIPATE IN) task -- with the generation step assumed. Figure 6 gives one such chain.[5]

$$\begin{tikzcd}
\text{((PARTICIPATE IN) ((SESSION CONSOLE): SOME))} \arrow[d] \\
\text{(GET (KNOW (ANSWER (WHETHER (CAN ((PICK UP) BLOCK-A))))))} \arrow[d] \\
\text{(ASK-AND-ANSWER (WHETHER (CAN ((PICK UP) BLOCK-A))))} \arrow[d] \\
\text{(UNDERSTAND (ASK (WHETHER (CAN ((PICK UP) BLOCK-A)))))}
\end{tikzcd}$$

Figure 6

5. To simplify the presentation, figure 6 is in fact a chain of calls to activities, not a chain of events. Events are slightly more complicated than this, since they are specialized by numbers to make them unique.

Here, it is assumed that *the block* refers to Block A in the current context.

In figure 6, note that the surface semantic representation is not present as an event on the event tree (although it can be associated with the UNDERSTAND event as a property). The interpreter level version of the question looks something like the surface semantic representation, but it is not identical. For other utterance forms, the divergence will be even greater. Consider, for example, some of the different ways to ask someone to write a program:

(1) I request that you write a program to manipulate blocks for me.
(2) Write a program to manipulate blocks for me.
(3) Would (will) you write a program to manipulate blocks for me?
(4) Could (can) you write a program to manipulate blocks for me?
(5) I want (need) you to write a program to manipulate blocks for me.
(6) I want (need) a program to manipulate blocks written.
(7) I want (need) a program to manipulate blocks.
(8) I would like you to write a program to manipulate blocks for me.
(9) I would like a program to manipulate blocks written.
(10) I would like a program to manipulate blocks.
(11) I request that you give me a program to manipulate blocks.
(12) Give me a program to manipulate blocks.
(13) Would (will) you give me a program to manipulate blocks?
(14) Could (can) you give me a program to manipulate blocks?
(15) I want (need) you to give me a program to manipulate blocks.
(16) I would like you to give me a program to manipulate blocks.
(17) Write me a program, would (will) you?
(18) Write me a program, could (can) you?
(19) Give me a program, would (will) you?
(20) Give me a program, could (can) you?

This is not necessarily a complete list, but it does give an idea of the number of request forms that must be handled.

This problem of relating utterances to their illocutionary forces has received a fair amount of attention in the linguistics literature. One approach is that given in <Gordon and Lakoff>. Concentrating primarily on requests, Gordon and Lakoff propose a set of four sincerity conditions; they then divide sincerity conditions into two sets: speaker-based and hearer-based. The following rule is then expressed:

> One can convey a request by (i) asserting a speaker-based sincerity condition or (ii) questioning a hearer-based sincerity condition.

Thus, with a single, powerful rule and a small number of sincerity conditions Gordon and Lakoff propose to account for the different ways that a request can be framed. There is some question, however, whether this rule is *too* powerful. For example, there is some disagreement about whether a sentence such as "I assume that you won't take out the garbage," which can be generated by the rule, is a legitimate request form. A critique in <Sadock> goes beyond the particular formulation found in <Gordon and Lakoff> to the possibility of accounting for the variety of ways to frame a request using sincerity/felicity conditions and a general rule. Our implementation bears more similarity to the ideas of Sadock than to those of Gordon and Lakoff with respect to the handling of the different surface realizations of an illocutionary force. Note that we also join with Sadock in distinguishing utterances that have an illocutionary force as their meaning and those that entail a given illocutionary force. For example, "It's cold in here." would not be a request to open a window but might entail such a request. Entailed illocutionary forces are not now handled by the system, but they could be handled by a program built on top of the mechanism that I will describe next.

To handle the different varieties of indirect illocutions, the Susie Software system contains a set of patterns corresponding to the different forms of top level lead-ins. These patterns have been called *keys*. Note that although the Susie environment is restricted to two top level activities (program writing and question answering), the patterns in the keys are more general than this; they match requests for information of any sort and requests for any sort of activity with the user as beneficiary. While the keys are special-purpose patterns, then, their usefulness is not restricted to the Susie Software environment.

Keys come in several different varieties. First, there are metadiscussion keys and top level lead-in keys. The former are discussed in section 10. Top level lead-in keys are specializations of the OWL relation, TL-LEAD-IN-KEY, and are of two types: terminal and non-terminal. Terminal keys are used to match a surface semantic representation against the actual generation step in a conversational method (e.g., the UNDERSTAND event in figure 6), while non-terminal keys are used to fill in the intervening events between the generation step and the console session event on the event tree (e.g., the GET and ASK-AND-ANSWER events in figure 6).

In general, context is not particularly helpful in anticipating the nature of requests for new tasks. (Cues probably do exist, but at this time it is not clear exactly what they are or how they should be used.) We have therefore adopted a bottom up scheme for matching against keys. The event tree path is found by starting with the step corresponding to the user's input and working up. I will discuss this matching process further below, but first let us take a closer look at the representation used for keys.

Non-terminal keys are somewhat simpler, so I will start with them. The following is an example of a non-terminal key, with case specifications omitted so that the structure will be clear:

```
[((TL-LEAD-IN-KEY NON-TERMINAL)
   ((COMMAND-AND-RESPOND ACTIVITY:) THE))

     <-- (GET ((BE DONE) (ACTIVITY: THE)))]
```

The different THEs above are used to make unique concepts where these are needed to keep the constraints straight for matching. The specializer of a non-terminal key relation is a substep of the method that matches the call found in the key's value. (Recall that OWL values are set using left arrows.) Alternatively, the value may be a call to a disambiguation routine; of this, more below. Thus, if the interpreter knows that it is in a COMMAND-AND-RESPOND event of the type given, it may assume, using this key, that the event immediately superior is a GET event of the type found in the value.

Non-terminal keys are links between two interpreter level representations; in addition, the interpreter still needs to be able to relate the user's utterance to the generation step that produced it. For this, terminal keys are used. The following is a terminal key example, again streamlined by the omission of case specifications:

```
[((TL-LEAD-IN-KEY TERMINAL)

(((SAY DECLARATIVE)
  (WANT-NEED [PRIMARY-SUBSTANTIVE:6
              (OBJECT (ACTIVITY:5 SOME)) <-- ::]))
                                     THE))

           <-- (ORDER-REQUEST (ACTIVITY:5 THE))]
```

The specializer of this key relation would match, among other things, the surface representation for the user's utterance, "I need a program to manipulate blocks written." Note that this specializer is not itself a surface semantic representation, but it is also not the interpreter level representation of the utterance -- (ORDER-REQUEST (ACTIVITY:5 THE)) is. I will call this type of representation a *subsurface* one. With the appropriate contortions by the reference finder/matcher, a surface semantic representation can matched against a subsurface level one. An identifying characteristic of subsurface representations is that they have one of three concepts as generalizers: (SAY DECLARATIVE), (SAY INTERROGATIVE), and (SAY IMPERATIVE). These correspond to whether the surface English form is declarative, interrogative, or imperative. As we can see from the example, surface form does not necessarily correspond with interpreter level form along this dimension; where the surface form of the example is declarative, the interpreter level form is ORDER-REQUEST, which is imperative in force.

While the specializer of a terminal key is a subsurface level representation, the value is always interpreter level (since in unambiguous cases this value corresponds to a method step). There are currently three interpreter level generation steps: TELL, ASK, and ORDER-REQUEST. These three are sufficient for the sample dialogue and probably for the Susie Software environment, but in a broader environment the set would have to be extended to include warnings, threats, promises, suggestions, and the other varieties of speech acts.

In the Susie environment, "I need a program to manipulate blocks written" is treated as unambiguous and the lead-in to a COMMAND-AND-RESPOND event. In many situations, of course, there will be more than one possible value for a key (either terminal or non-terminal), and, when this is the case, the key's value will be a call to a special OWL disambiguation procedure. All of the routines that disambiguate for method invocation belong to the class DISAMBIGUATE. The fact that disambiguation routines are coded in OWL means that they can be examined and explained by the interpreter. This is part of the general policy that takes as many choices as possible out of black boxes, allowing inspection and evaluation. It also means that the reasoning processes needed to do disambiguation can be done by the OWL reasoning module Whether.

Note that the structure of keys shown here is the form produced by the programmer. When the keys are loaded into the system, the OWL reader adds them to the concept tree, automatically creating a key subtree which can be used in matching. (See <Hawkinson>.) When a surface representation is output by the parser, the general scheme is for the matching algorithm to start at (TL-LEAD-IN-KEY TERMINAL) and move down specialization links until a match between the surface representation and the key specializer is found.

After matching against the specializer of the terminal key, the interpreter has either the procedure call found in the terminal key's value position or a call provided by the evaluation of a disambiguation procedure. This call can then be used to match against specializers of the relation (TL-LEAD-IN-KEY NON-TERMINAL) in the same way that was done for the terminal keys. The value of the non-terminal key found would be used in the next non-terminal key match, and so on until a path from the event corresponding to the user's utterance to Susie's top level routines had been constructed.

Remember that keys are used only for top level lead-ins. Thus, not all procedures and utterance types will have associated keys -- just those that lie on an event tree branch below a top level activity. Although the set of top level goals may eventually be extended, there will still remain substeps and procedures without associated keys. Thus, one justification for using a special structure rather than a more general search procedure is that top level lead-ins are a subset of the possible lead-ins to conversational methods. Using this special structure, the interpreter will not try to construct paths that are known *a priori* to lead to deadends. Similarly, a general bottom-up search mechanism that tried to construct a path from lead-in to top level activity would be slowed down considerably by the fact that lead-ins are not necessarily produced by the first step in a method. (Recall that they are produced by the first step executed in *recognition mode* when Susie is not the AGENT of the method.) Given the way that OWL method steps are linked, the fact that a path could not be reliably constructed from first steps is bound to make a general search mechanism inefficient.

Keys are used to enumerate the different utterance types possible for use as top level lead-ins, but they do not give criteria for choosing between them. In fact, another OWL structure is responsible for representing these criteria: the WAY evaluator. This structure will be discussed in the next section. Once the correct event tree path has been found, the interpreter can match against WAY evaluators to derive the implications of a choice.

The last topic for this section is whether top level lead-in matches must always be attempted. Are there types of inputs or dialogue contexts for which matches against keys are clearly ruled out? First, for typed dialogue it appears that for two sorts of input the top level lead-in possibilities need not be tried at all:

(1) Sentence fragments, including placeholders such as *O.K.*

(2) Complete sentences beginning with *yes* and *no*[6]

For these two categories of user input, there is no need to match against keys. Second comes the question of whether there are some contexts in which top level lead-ins do not occur. I think that for most people, a dialogue involving two semantically unrelated program writing tasks would be either too complex or very close to it unless the processes were synchronized (i.e., both problem statements given in sequence, then both initial descriptions, etc.). Similarly, when a question is asked by a person in a dialogue situation it would be considered impolite to ignore it and talk about something else altogether without first acknowledging the question in some way. For people, then, there seem to be some contexts where top level lead-ins are unlikely.

Whether or not this is true for person-to-person communication, I suspect that interaction with a computer changes the rules of dialogue significantly. The computer does not have the same memory limitations as a person, so that if a user can manage to start off a new task in the middle of an old one and keep them both going at the same time, then this will probably come up in the course of Susie Software console sessions. Again, the goals of a computer do not have to be respected in the same way that the goals of another person do, so that we can expect the user to occasionally ignore questions, requests, etc. without bothering to acknowledge them, going on to initiate a new task. The upshot of this is that top level lead-ins should be considered to be possible, irrespective of the current context.

6. The only exception to (2) that I can find is where yes is used to establish a helping or command relationship (e.g., "Yes, I'd like to know when the 5:15 train gets in to Portland."). This seems to be in answer to an assumed "Can I help you?" This special case should be easy to screen, since it is always found at the beginning of the conversation after the hello-ing is done.

## 8. Fitting User Input to Open Tasks

Susie Software uses the conversational methods to generate structural expectations that are dependent on the course of the console session to date. In this section I will consider the situation in which execution of one or more methods is already in progress and the next step on a standard path is to be executed in recognition mode. To perform the appropriate match, the interpreter first needs the ability to detect possible next steps, and, second, it needs to do the actual matching as efficiently as possible. These two topics are discussed in the subsections that follow.

### 8.1. Expectations

For a given step in a method, how many possible standard path successor steps are we talking about? There is, of course, the possibility of branching within a method. In addition, with the existence of the conditional statement IF-THEN or any other mechanism that permits a step to be optional, the interpreter must be ready to recognize either the optional step or its successor. Another complication arises from the fact that the end of some steps is not always clearly marked (e.g., in giving a description). The interpreter must be ready, then, to detect the completion of a step by matching against steps that continue methods higher on the event tree.

This seems to be about the extent of the possibilities, since there are some sorts of situations that probably do not come up. In the specifications for Susie Software, we admit the possibility of carrying out two activities (e.g., writing two programs) by alternating steps. To switch from one activity to another and back, however, it appears that the rules of dialogue require the use of metadiscussion. The user would have to specify which task a particular utterance applied to. Similarly, to re-open a previously completed or suspended activity, the user would have to state his intentions explicitly. Neither of these situations, then, would have to be handled by the normal standard path expectations mechanism. We therefore come up with a relatively small number of possible standard path successor steps; in my experience, five has been about the limit, and ten probably borders on pathological for this environment.

With such a small number of possible standard path successors, how could we possibly run into trouble? Unfortunately, there is not far to look. Successor steps may be calls to other OWL procedures, so that, given a step, the user's input may not match the successor, but instead may match a *substep* of the method that matches the successor. For example, when Susie asks the user for a description of the input and output conditions of his desired program, she executes the step (ORDER-REQUEST (DESCRIBE OBJECT:):I). The recognition mode step that follows, however, is not a TELL, but instead, a call to STATE-AND-ACKNOWLEDGE which itself contains a call to TELL. In general, it may be necessary to go through several layers of calls and procedure selection processes before the actual TELL step is encountered.

The search for a basic generation step (currently, TELL, ASK, or ORDER-REQUEST) is a potentially expensive operation, since there may be more than one method matching a call, and the method selection process itself is a relatively expensive one. The dialogue system has gone through several different implementation phases in an attempt to deal with this problem well. The current scheme is a mixture of top-down and bottom-up matching that relies on two assumptions about constraints on the dialogue processing environment. First, the scheme uses the assumption that the distance between an expectation and a basic generation step is no more than two events, and, second, it relies on the existence of a bounded, relatively small, and predictable set of conversational methods to carry out a given step.

The matching scheme for non-terminal expectation steps uses relations associated with those conversational methods which may start off with recognition steps (or assumption steps followed by recognition steps). An example of such a relation follows:

```
[(LEAD-IN
  (GET ((BE DONE) [(ACTIVITY: SOME)
                   NONVERBAL])))
  <--
   (OR (ORDER-REQUEST (HELP (ACTIVITY ASK-AND-HELP)))
      (ORDER-REQUEST (ACTIVITY COMMAND-AND-RESPOND))))]
```

This says that, for the steps of the GET that have verbal recognition substeps, there are currently two possible types of lead-in. (There are, however, a number of ways to phrase the lead-in; this matter is considered below). The LEAD-IN relation gets us from the non-terminal call to all possible lead-ins, so that the interpreter will be able to detect non-matches quickly and can move on to other alternatives.

Once such a LEAD-IN relation is matched, the interpreter can construct an event for the basic generation step (in this example, one of the ORDER-REQUESTs) and an event for its containing activity, which is recoverable from the specializer of any variables in the LEAD-IN's value. For the example, the containing activity is either COMMAND-AND-RESPOND or ASK-AND-HELP. Once events for the basic generation step and the containing activity are constructed, an attempt is made to attach them to the event tree. The current assumption, mentioned above, is that the top of the event pair will either match the expectation directly or will be an immediate subevent of it.

Note that for the sample dialogue I have already run into situations where this assumption is too restrictive, and I have wanted to write methods with more calling depth. There are several different alternatives here. It may be that the restriction on depth can be relaxed somewhat without a complete sacrifice of efficiency. This may result in a new limit on calling depth that does not represent an undue constraint on the OWL programmer. I plan to experiment with this alternative. For the present, and if relaxing the assumption proves unworkable, there is the alternative of flattening out methods. For the conversational methods, there seems to be no difficulty in taking out some of the intervening calls and substituting the actual substeps instead. The main justifications for the intervening calls handled so far were conceptual clarity and avoiding the duplication of code. If the flattened method alternative is chosen, in the long run, the system can be made more amenable to programmers by the addition of an OWL macro facility, allowing macros to be assigned concept names. This would allow the OWL programmer to structure the methods as desired but would still guarantee that calling depth was constrained enough for efficient recognition matching.

One problem that is always with us is ambiguity. For standard path successors, at least within the framework of the current matching scheme, I do not believe that we will find disambiguation mechanisms to be as important as they are elsewhere in the system. With a stop-on-success matching scheme, ambiguities among possible standard path successors are not even detected, and, where one occurs, the system must rely on the user to detect any errors made. In a try-all-possibilities scheme, on the other hand, disambiguation plays a more prominent role theoretically, in that more than one standard path successor could match an utterance. But, even in this case, it appears that our domain is structured in such a way that few ambiguities will occur between standard path possibilities and most will occur across types (e.g., one top level lead-in and one standard path successor). At any rate, to the extent that standard path ambiguity is detected the remedy would seem to be the use of a few general purpose disambiguation procedures (such as asking the user for more information) rather than depending on special purpose methods. This is because we will not in general be able to predict the sorts of ambiguities that will occur between standard path successor steps in the same way that we can for, say, top level lead-ins.

## 8.2. WAY Evaluators

Identifying possible successor steps is not, of course, enough. Just as for top level lead-ins, there will tend to be several different ways to phrase a next-step utterance. To access these different utterance forms, we can use a method type that I have called an *evaluator*; the similarity between this name and the Evaluate module is intended. An evaluator is an OWL routine with a header that is a relation and a standard path that gives criteria for choosing between possible values. The evaluators that are used here are WAY evaluators, and they give alternate ways to convey the same interpreter level message. The use of WAY evaluators will be explained here for recognition, but they are also intended to be used for generation.

An example of a WAY evaluator can be found in figure 7.2, preceded by an English translation in figure 7.1. Note that I have omitted the case assignents for the PRINCIPAL-RESULTs to make the OWL version more compact. This particular WAY evaluator is applied in situations where a statement has just been made and the acknowledgment given is meant to convey that the statement has been understood. Among the ways to phrase the acknowledgment, as indicated in the PRINCIPAL-RESULTs, are, "O.K.," "I understand," "I see," and "I get it."

```
[find a way to acknowledge a statement
   object: a variable for the ways to acknowledge a statement
   agent:  a person or computer system

   method:

      either
        (1) If the context is informal and it is a routine process
            to integrate the statement in with existing knowledge,
            say "O.K."

        (2) If the context is informal but the integration process
            was not routine, say "I get it."

        (3) If the context is formal and the integration process
            was quite difficult, then say "I see."

        (4) If the context was formal and the integration process
            was not routine, then say "I understand."]
```

Figure 7.1

```
[(EVALUATE (WAY (TELL (ACKNOWLEDGE TELL))))
  PLAN
  OBJECT:
   (WAY [(TELL [(ACKNOWLEDGE
             [(TELL SUMMUM-GENUS:)
                AGENT::::      <--  (OR HUMAN:2 VERBALIZER:2)
                DESTINATION:::: <--  (OR HUMAN:1 VERBALIZER:1)])
             AGENT::: <-- (OR HUMAN:1 VERBALIZER:1)])
      AGENT::       <-- (OR HUMAN:1 VERBALIZER:1)
      DESTINATION:: <--  (OR HUMAN:2 VERBALIZER:2)])
  AGENT: <--  (OR HUMAN:1 VERBALIZER:1)
 METHOD: <--
 (OR
  (IF-THEN (AND ((BE INFORMAL) (CONTEXT CURRENT)):1
           ((BE ROUTINE) (INTEGRATE
                ⋄TELL2=
                (TELL (SUMMUM-GENUS: THE)))))

      (BECOME (PRINCIPAL-RESULT :
                ((SAY DECLARATIVE) OK):)))

  (IF-THEN (AND ((BE INFORMAL) (CONTEXT CURRENT)):2
           (((BE ROUTINE) (INTEGRATE ⋄TELL2)) NOT))

      (BECOME (PRINCIPAL-RESULT :
           ((SAY DECLARATIVE)
            ((GET IDEA) (PRINCIPAL-RESULT ⋄TELL2)?)))))

  (IF-THEN (AND ((BE FORMAL) (CONTEXT CURRENT)):3
           ((BE DIFFICULT) (INTEGRATE ⋄TELL2)))

      (BECOME (PRINCIPAL-RESULT :
           ((SAY DECLARATIVE)
            ((SEE CONCEPT) (PRINCIPAL-RESULT ⋄TELL2)?))))

  (IF-THEN (AND ((BE FORMAL) (CONTEXT CURRENT)):4
           (((BE ROUTINE) (INTEGRATE ⋄TELL2)) NOT))

      (BECOME (PRINCIPAL-RESULT :
           ((SAY DECLARATIVE)
            (UNDERSTAND (PRINCIPAL-RESULT ⋄TELL2)?)))))))]
```

Figure 7.2

There are no doubt more possibilities and the conditions for each could be tightened up, but this example is enough to sketch out the approach that has been taken. What the WAY evaluator offers is a link between certain interpreter level representations and a set of subsurface level forms. In its primary use the WAY evaluator would be run as part of the generation process, to select a basic subsurface semantic form for an ASK, TELL, or ORDER-REQUEST call in the current environment. The phrase *in the current environment* is crucial because it justifies the use of a procedural form; otherwise, a simple list of the possibilities would be sufficient.

For recognition mode matching, the interpreter will be starting with an ASK, TELL, or ORDER-REQUEST step. Using this step, the recognition module constructs an OWL activity call, then uses the interpreter's method selection routine to choose an appropriate WAY evaluator. The PRINCIPAL-RESULT alternatives of the evaluator found can then be used to match against the surface representation of the user's input. Either one subsurface level alternative is found to match or, if none do, then the successor step can be eliminated as a possibility altogether. Once a match is found, the implications of a user's choice can be derived by an inspection of the path of tests leading to that particular PRINCIPAL-RESULT. While these implications are not particularly important for our purposes right now (since many center on politeness and authority relationships), they are important in human-to-human interaction and might prove useful as we try to fine-tune the system.

WAY routines are limited in their uses and simple in their structure, but their style is important. They are special-purpose structures to perform a function that might be done by a general deduction mechanism in other systems. By differentiating among the sorts of inferences that must be made, we can isolate special bits of knowledge and know exactly when they are to be accessed.

## 9. The Best-Laid Plans: Recognition in Failure Situations

Our next topic is what happens when a failure situation either occurs for the conversational partner or is detected by him. He reports the failure, and it is the job of the interpreter to recognize this report. How can this be done? I will first discuss the access of expectations for failure conditions and then look at matching.

The access of failure expectations will differ according to whether the system is looking for recovery paths or autonomous OWL failure methods. As I noted above, autonomous failure methods are not needed for the sample dialogue, so this area of the design is not well developed. Existing mechanisms could be used to access and match these failure handlers, but it is not clear that this will be adequate as the number of failure handlers grows. It seems better to delay any speculation about the sort of special mechanism that would be required until direct experience with this sort of situation is gained.

This leaves the problem of recognition of lead-ins to recovery paths. Access of recovery paths associated with a step is the same for recognition as for carry-out mode. Recovery paths are so specific that there will not be a great number of them for any given failure, so that each possibility can be accessed and tried in turn. One feature of recovery paths that was mentioned in passing above is that they may be associated with higher level methods; therefore, the particular recovery paths available for a given failure become context dependent. This feature causes no special difficulties for recognition, however, and the carry-out routines that access a recovery path possibility can be used for recognition as well.

One special situation does come up due to the possible presence of steps executed in assumption mode. For example, Susie asks a question. The user then tries to understand the question and finds the answer. His next standard path successor step would be to give the answer, and this is Susie's first recognition-mode step. If, however, something has gone wrong, then the user's next utterance may be part of a recovery path. The failure could have occurred in the understanding process (e.g., "I don't know what you mean by..."), in the answer finding process, or could be related to the process of giving the answer. Thus, it is necessary to check recovery paths related to assumption steps as well as recognition steps. The order in which these sets of step-related paths should be considered is not clear. Until we have additional experience, the natural order in the methods is being used, since this seems to be as good as any.

The next topic for this section is matching. Basically, it looks like recovery path lead-ins should be handled in a manner similar to standard path successor steps, i.e., by matching on PRINCIPAL-RESULTs of WAY evaluators. The mechanism used for handling non-terminal standard path successor steps has also been adopted here.

The only difference between standard and recovery path recognition steps seems to lie in the WAY evaluators themselves. Some utterance types seem exclusively failure related, such as the statement of a lack of information as a way to request it (e.g., "I don't know X.") Moreover, the connotations of some utterances seem to differ according to whether thay are used as recovery path lead-ins or in other contexts such as top level lead-ins. These two facts together point toward independent WAY evaluators for recovery paths. These look like the evaluator in figure 7 of the last section, with the exception that the header concept contains an AND with an explicit mention of a failure call. For example,

```
[(EVALUATE (WAY (AND (TELL FAIL):
           (ASK QUESTION:))))
  ...]
```

This would be the initial part of a procedure evaluating the different alternatives for asking for information in a failure situation.

Finally, once we have launched into a recovery procedure, what are the possibilities for the next utterance typed in? I suspect that it is not possible to go back to the standard path without either an explicit transition step in the recovery procedure or metadiscussion.

This winds up recognition for failure situations. Basically, recovery paths access will be similar to access in carry-out mode, and matching will be similar to that done for standard path successor steps.

## 10. Metadiscussion

Metadiscussion recognition is done using a set of patterns that I will call *metadiscussion keys*. These are similar to top level lead-ins keys, except that metadiscussion keys are not a summary of existing structure in the same way that top level lead-in keys are; instead, they are the structure.

In the Susie Software environment, three sorts of metadiscussion seem to be possible: either an activity is suspended, a previously suspended or completed activity is reopened, or a description of what is to come next is given. Accordingly, the metadiscussion keys are specialized by either SUSPEND, RESUME, or INFORM. An example of a metadiscussion key for resuming an activity is given here, with most of the semantic case specifications omitted to streamline the presentation:

```
[((METADISCUSSION-KEY RESUME)

 (((SAY DECLARATIVE)
     [(WANT
        (FINISH ACTIVITY:))
       TIME-RELATION:: <-- NOW])
                      THE))

    <--  (RESUME (ACTIVITY: THE))]
```

The specializer of a metadiscussion key is a subsurface level representation against which the surface level representation output by the parser will be matched. This particular example would match an input such as "I want to finish writing the tower program now." The value of a metadiscussion key is an interpreter level call to an activity, in this case a call to RESUME. When the key specializer is matched, the ACTIVITY: variable is bound to the interpreter level representation of the activity that the user wants to restart (e.g., the referent of *the tower program*). The call can then be matched by the carry-out method selector, and Susie can reopen the previously suspended or completed activity.

There is one potential difficulty with the use of metadiscussion keys. Not all metadiscussion is sentential. Some seems to be phrasal, as in *By the way*, *Wait a minute*, etc. These phrases themselves are easy enough to recognize, and they lend themselves well to screening, but it also appears that time phrases are used as metadiscussion. Consider the following example from <Deutsch>:

A: I have the jaws around the hub. How should I take it off now?

E: Tighten the screw in the center of the puller...that should slide the wheel off the shaft.

A: OK. It's off.

A: A little metal semicircle fell off when I took the wheel off.

Here, the phrase *when I took the wheel off* seems to be metadiscussion associated with the reopening of a task (to initiate a recovery procedure) after the task was implied to be finished. But time phrases also appear in what I would judge to be non-metadiscussion situations, e.g., "After you pulled the plug, did the water run out?". Here, the time phrase is used to select semantic context rather than to indicate the flow of control. At least where time phrases are involved, then, screening for metadiscussion may not be a trivial operation, and the rest of the utterance may have to be examined with care. Although this seems to be the case, my best estimate at this point is that checking for metadiscussion will still be the cheapest recognition process and should be done before other matches are attempted. Metadiscussion keys can form the backbone of the matching scheme, augmented by additional mechanisms to handle the more difficult subclasses.

## 11. Conclusions

This paper has outlined a framework for processing mixed-initiative typed dialogue, with special attention to recognition. Recognition is done using a set of conversational methods written in OWL, a set of special recognition patterns, and a mixed matching strategy. The conversational methods and the recognition patterns are used to provide structural expectations, some of which are developed dynamically. The task-oriented nature of the environment means that the structural expectations will be a relatively good source of information, and this in turn allows a good deal of flexibility to be incorporated into the Susie Software system.

The important question for a natural language system of this sort is its extensibility. Can the design presented here be adapted for real-world interaction with an expert system? Since the implementation is still in progress, I cannot answer this question at this time, but I can list some of the ways that the need for extensibility has been anticipated in the design.

First, the conversational methods range from semantic domain dependent plans (e.g., the program writing one) to totally domain independent ones, e.g., ASK-AND-ANSWER. If we were to add new tasks to the Susie Software environment the more general conversational methods would continue to be applicable. Moreover, taking Searle's speech acts as a guide, it is probably possible to write a complete set of domain independent conversational methods. (I would estimate that the number of these would be on the order of a hundred and certainly less than a thousand.) Beyond the accompanying recognition patterns, no new structure would have to be added to the system to accomodate these new conversational methods, and at one level we could then say that we had a very general system on our hands.

There is more to dialogue, however, than domain independent structures, and the question of extensibility is more problematic *within* the domain that has been chosen. In order to provide users with a working system, we will have to span the semantic domain. This requires more domain dependent conversational methods, which are relatively straightforward. Beyond this, it requires special structures and a good deal of built in knowledge to handle reasoning, reference resolution, and the framing of messages for generation. Crucial to this is also a facility for modelling the knowledge of the user. Space considerations have constrained the attention I could give to these areas here; however, they have not been ignored in Susie Software development. While spanning the semantic domain constitutes the next challenge in development of the system, we have developed what we believe to be a firm groundwork in reasoning, user modelling, etc. I hope to discuss some of these areas in forthcoming papers, as well as focus on them in the course of further development of the system.

## References

Bruce, Bertram C., "Belief Systems and Language Understanding," BBN Report No. 2973, Bolt, Baranek and Newman, Inc., January 1975.

Carbonell, Jaime, *Mixed-Initiative Man-Computer Instructional Dialogues*, Doctoral thesis, Electrical Engineering Dept., M.I.T., 1970.

Deutsch, Barbara G., "The Structure of Task Oriented Dialogs," Technical Note 90, SRI Project 1526, Stanford Research Institute, April 1974.

Gordon, David and Lakoff, George, "Conversational Postulates," *Papers from the Seventh Regional Meeting of the Chicago Linguistics Society*, 1971.

Hawkinson, Lowell, "The Representation of Concepts in OWL," *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, September 1975.

Marcus, Mitchell, "Diagnosis as a Notion of Grammar," *Theoretical Issues in Natural Language Processing*, An Interdisciplinary Workshop in Computational Linguistics, Psychology, Linguistics, and Artificial Intelligence, Cambridge, Mass., June 1975.

Martin, William A., "A Theory of English Grammar," Technical Memorandum, Laboratory for Computer Science, M.I.T.

Ogden, C.K., *Basic English*, New York: Harcourt Brace and World, 1968.

Sadock, Jerrold M., *Toward a Linguistic Theory of Speech Acts*, New York: Academic Press, 1974.

Searle, John R., *Speech Acts*, Cambridge: University Press, 1969.

Sunguroff, Alexander, "OWL Interpreter Reference Manual," Automatic Programming Group, Laboratory for Computer Science, M.I.T., October 1975.

Winograd, Terry, *Understanding Natural Language*, New York: Academic Press, 1972.

Woods, William A., "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, 13, October 1970.