ARTIFICIAL COMMUNICATORS: AN OPERATING SYSTEM CONSULTANT

BY

PAUL MC KEVITT, B.Sc. (Hons.)

A Thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the Degree

Master of Science

Major Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

May 1988

‘‘Artificial Communicators: An Operating System Consultant,’’ a thesis prepared by Paul Mc Kevitt in partial fulfillment of the requirements for the degree, Master of Science, has been approved and accepted by the following:

____

William H. Matchett

Dean of the Graduate School

____

Yorick Wilks

Chairman of the Examining Committee

____

Date

Committee in charge:

Dr. Yorick Wilks, Chairman

Dr. John Barnden

Dr. Don Dearholt

Dr. Fred Richman

Dr. Roger Schvaneveldt

Tiomnaĭtear an tráchtas seo do Pheadar, Róise, Peadar Og, Tara, Micheál agus a lán daoine in Eirinn agus in áiteanna nach ĭ ar bhuail mé leo i rith mo shaoil. Is mar gheall orthu siúd a spreagadh mé chun an taighde seo a dhéanamh.

This thesis is dedicated to Peter, Rose, Peadar, Tara, Michael and many people in Ireland and elsewhere that I have met throughout my life. Research and ideas have been stimulated by those around me from the early years until now.

ACKNOWLEDGEMENTS

This thesis∗ was developed in the exciting surroundings of the Computing Research Laboratory† (CRL) under the direction of Dr. Yorick Wilks. The work herein has been presented to the Natural Language and Knowledge Systems Groups at the CRL and audiences at international conferences. Computer Scientists, Psychologists, Linguists, Philosophers and Mathematicians have made various interesting comments. They are all to be thanked for suggesting loopholes and further work.

The project was initiated when Yorick Wilks asked me to hook up a natural language parser and a UNIX‡ knowledge base. Little did he, or I, know that the project would grow, from just being a quick parser-database interface, into a theory on how to construct a natural language understander for an operating system consultant. The project has developed into a theory which was presented, under invitation, at the *First International Workshop on Knowledge Representation in the UNIX Help Domain*, held at the University of California, Berkeley, in December 1987.

I thank Yorick Wilks for being a constant guide in the completion of this theory. He has been an excellent promoter of this work and has always provided the utmost encouragement. Thanks are also due to Derek Partridge who continually showed an interest in the research and gave me an opportunity to publish a paper in his latest book on *Artificial Intelligence and Software Engineering (Vol. 1)* from

∗ The research herein is currently under proposal to the Office of Naval Research (ONR), the Air Force Office of Scientific Research (AFOSR), Rome Air Development Center (RADC), AT&T, and U S West Advanced Technologies. They are all to be thanked for considering this work for funding.

† The Computing Research Laboratory is a Center of Technical Excellence partially funded by the New Mexico State Legislature as part of the Rio Grande Research Corridor. This work was supported by a special research assistantship from the Computing Research Laboratory.

‡ UNIX is a trademark of AT&T Bell Laboratories.

Ablex Corporation.

I thank my committee for providing interesting discussion and argument on my thesis. They suggested possible problems on a closing draft of the thesis. John Barnden, Roger Schvaneveldt, Fred Richman and Don Dearholt are certainly to be commended for doing that.

The theory herein has been scrutinized by the Natural Language Group at the CRL on many occasions. Members of that group, including Yorick Wilks, Jerry Ball, Afzal Ballim, Dan Fass, Tony Plate, Cheng-Ming Guo, Xiuming Huang, Brian Slator and Sylvia Cand. De Ram, have constituted a stimulating environment during the pursuit of this research. Special thanks are due to David Farwell who guided me from many pitfalls and mineshafts. Mike Coombs, Roger Hartley, Chris Fields, Dan Eschner, and Eric Dietrich from the Knowledge Systems Group made interesting comments when the work was presented to them. I have worked closely with Stephen Hegner at the University of Vermont and he has provided enlightening suggestions.

I commend New Mexico State University (NMSU) for providing me with the excellent interdisciplinary research environment of the Science Building encompassing the Departments of Computer Science, Psychology and Mathematics. I also thank NMSU for recognizing this research through awarding me the *Young Centennial Researcher Award* in this Centennial Year of Celebration.

VITA

January 9, 1964 — Born at Monaghan, Co. Monaghan, Republic of Ireland

1985 — B.Sc. (Hons.) in Computer Science, University College Dublin,
The National University of Ireland (NUI), Republic of Ireland

1986-1988 — Research Intern at the Computing Research Laboratory

Publications

Mc Kevitt, Paul (1988) "Artificial Communicators: An operating system consultant," *Master's Thesis*, Department of Computer Science, Dept. 3CU, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul & Wilks, Yorick (1987) "Inference rules in an operating system consultant," In *Preprints of the First International Workshop on Knowledge Representation in the UNIX Help Domain*, University of California, Berkeley, California, December.

________ (1987) "Transfer Semantics in an Operating System Consultant: The formalization of actions involving object transfer," *Proceedings of The Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Vol. 1*, 569-575, Milano, Italy, August.

Mc Kevitt, Paul (1987) ‘‘Natural language interfaces in computer aided instruction — What happened before and after the 80s AICAI coup,’’ *Proc. 4th International Symposium on Modeling and Simulation Methodology*, University of Arizona, Tucson, Arizona, January.

________ (1986) ‘‘Formalization in an English interface to a UNIX database,’’ *Memoranda in Computer and Cognitive Science, MCCS-86-73*, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

________ (1986) ‘‘Building embedded representations of queries about UNIX,’’ *Memoranda in Computer and Cognitive Science, MCCS-86-72*, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

________ (1986) ‘‘Selecting and instantiating formal concept frames,’’ *Memoranda in Computer and Cognitive Science, MCCS-86-71*, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

________ (1986) ‘‘Object and action frames for Transfer Semantics,’’ *Memoranda in Computer and Cognitive Science, MCCS-86-69*, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Presentations

- Colloquia of Natural Language Work at the Computing Research Laboratory, October, 1987.
- Knowledge Systems Group Seminar at the Computing Research Laboratory, September, 1987.
- The Second Western Expert Systems Conference (WESTEX-87), Anaheim, California, July, 1987.
- Meeting with Dr. Steve Hegner at the University of Vermont, November, 1986.
- AT&T on their visit to New Mexico State University, October, 1986.
- Natural Language Group Seminars at the Computing Research Laboratory, 1986 and 1987.

Fields of Study

Major Field:

Computer Science, Operating Systems, Artificial Intelligence, Cognitive Science, Planning, Inference, Knowledge Representation, Natural Language Processing, Beliefs.

Abstract

Artificial Communicators: An Operating System Consultant

by

Paul Mc Kevitt, B.Sc. (Hons.)

Master of Science in Computer Science

New Mexico state University

Las Cruces, New Mexico, 1988

Dr. Yorick Wilks, Chairman

ABSTRACT

Operating systems are computer programs which allow people to accomplish various tasks on computer hardware. Many people find it difficult to learn how to use computer operating systems. In the past, help has been available from manuals or computer experts. However, manuals are very tedious to search through, and computer experts are scarce. It is possible to build computer programs which communicate with people on various domains. We develop a theoretical design for a computer consultant which communicates with people in English. The consultant will answer English questions on operating systems. The operating system consultant embodies in its design theories of knowledge representation, inference, planning and natural language processing.

Table of Contents

List of Figures

Chapter 1: Introduction

Artificial Intelligence (AI) is a field concerned with creating computer programs that exhibit intelligent behavior. One way in which a program may demonstrate intelligence is to act as a consultant on some topic. Such consultant programs can communicate with people through the medium of speech or natural language. To be useful any such program should approximate a real consultant as best as possible. Effective communication between the program and people will depend on how good that approximation is.

1.1. Learning operating systems

In everyday life computer users have to use new systems, or utilities on existing systems to accomplish various tasks. Many people find it hard to learn how to use operating systems. People at various levels of computer education may have a good idea of how to accomplish a task, but not the necessary specific knowledge needed to complete the task. It is often necessary for the user to obtain further information about the system or utility.

Information can be provided by manuals, on-line help, or computer experts. Experts on computer systems are in short supply and cannot always be available to help others with various problems. Often experts are not available at the time they are needed (see Mc Kevitt, 1987, p. 2). Documentation can be made available but such documentation is usually very large and cumbersome. Finding relevant information in a short period of time is quite difficult. Manuals are not always useful as they can supply numerous pages on anything related to a single command. Manuals will not always supply, in a single page, the answers on how to execute multiple processes, which involve the union of complex constraints. Often people do not know the command they are looking for, and therefore cannot index relevant information in the manual. If someone doesn't know how to use the system he may take up valuable time of other users by asking questions about how to do something. There is a definite need for computer programs which can communicate with a user and answer questions about the use of operating systems.

Computers can provide help by on-line documentation where relevant information can be located by simple key-matching routines. This is an upgrade of help provided by documentation manuals. However, there is another approach. What if a computer program could provide answers to queries and act like a real consultant? The computer program would be less expensive than an expert, and would be readily available. We call such a program an *artificial communicator*.

1.2. The essence of artificial consultants

We have now decided that computer programs are a useful tool in communicating information to the user. The next question we must ask ourselves is, ''What kind of program makes a good consultant?''. We could build a program which accepted single words like *delete* and gave answers like, ''To delete files or directories you do the following..''. That would be a simple keyword interface. On the other hand, the program could present a menu on the screen with the user choosing various options and indexing information about the required command. Such consultants are called menu-based interfaces.

However, there are some problems with menu-based interfaces. People can take a long time to work through the menu interface to find exactly what they are looking for. Even more problematic is the stark fact of nature that people always know what they want to do in some system, but can only express that in English or some other natural language.

We could build a program which accepts English queries and replies to these in English. It may take a while to build a program which accepts variations of natural language input but that would be a good solution. We call such consultant programs natural language understanders. This thesis is about the theoretical design of a natural language understander for an artificial communicator. The communicator is an operating system consultant.

Say we decide that the program will have an interface which accepts input in the form of English queries, ‘‘What else does the consultant need?’’ The program needs to know something about the subject on which it is meant to consult. The subject is operating systems. Therefore, all that needs to be done is to type in the very manuals that people have traditionally searched through. Yet, it isn’t as easy as that, because there must be some theory of how the manual should be represented in the system. In other words, ‘‘What is the best way to represent knowledge about operating systems in a consultant?’’, must be answered. Also we must find out the best way to select particular knowledge relevant to a particular question, and even more important is the problem of understanding natural language input.

If we build a program with an English interface which understands queries, by selecting the necessary knowledge to do that, then the problem of answering each query must be faced. To cut costs the knowledge that was used for understanding a query could be used to answer it. Yet, that isn’t the way to do it, because the problem of understanding a query is not the the same as that of answering. The answering process needs to know more detail than the understanding process. The answering process needs to know the constraints on some process to be executed by a user, and more important, which commands and options will perform that process. This has been pointed out by Hegner (1987, p. 1).

Any program with a natural language interface, a knowledge representation for understanding queries, and a knowledge representation for answering them should act as a good consultant. A module which answers queries in English would be added. Then we are done.

If we can build a program which accepts English queries, understands the queries, and answers them in English, then the program can act as a good consultant. Of course, many details of each component would have to be worked out, and good theories of design for each module would be beneficial. We can't go wrong if the program embodies a good theory of communication. To polish the communicator off, other modules would be added to track the context of conversation with the user, and to represent user beliefs about operating systems. A planning module would be added to understand plans appearing implicitly in queries, and to check if such plans were viable.

We have seen some of the problems that need to be investigated in order to build an operating system consultant which will act as an effective communicator. The consultant approach brings forward many of the major problems in AI research. AI research topics that come to mind are natural language understanding, knowledge representation, inferencing, planning, belief representation, natural language generation, user modeling, misconception detection and many more. Although we will touch on many of these topics in this thesis we do not intend any deep discussion on AI approaches to each. That would take up a lot of time. However we will take an approach on various problems, justify the approach, and compare that approach to other ones in the field.

1.3. Objectives of the thesis

The major theme of this thesis is to build a theoretical design of a natural language understander for an operating system consultant. A good feel for the problems involved in doing this is supplied by observing the types of queries people ask. A theory on query understanding will need to cover many types of query. Shown in Figure 1.1 is a list of some common user queries

---

"How do I print a file on the Imagen?"

"How do I print a file with pageheaders and line numbers?"

"What is a directory?"

"What is a file?"

"What is read protection?"

"How do I print a device file which has pageheaders?"

"What is the permanent storage limit?"

"What is the option on the cat command which numbers lines?"

"How do I print a listing of my directory on the laser printer?"

Figure 1.1. English queries on operating systems.

which the theory described in this thesis will cover. We intend to justify the theory by showing how it is used to understand various queries.

Most existing consultant systems for operating systems do not include natural language understanders. The few that do embody natural language understanders do not encapsulate strong formal models of operating systems. The originality of this work lies in the fact that we have developed a formal model and applied that model in a natural language understander. We have separated out the knowledge needed for understanding natural language queries from the knowledge needed to answer them. A knowledge representation, planning mechanism, and theory of meaning representations has been developed. We have not implemented much of the theory although there is an implementation of a plan understanding component called *Plan-Con*.

1.4. Organization of the thesis

The organization of the thesis is as follows:

- Chapter 2 contains a discussion on the problems involved in building operating system consultants and how others have tackled them. We characterize different approaches showing the advantages and disadvantages of each.
- Chapter 3 proposes a theory of how knowledge about operating systems may be represented in a computer program. The knowledge representation is called *Transfer Semantics*.
- Chapter 4 introduces a formal language for describing commands or actions and various manipulations of knowledge structures. The language is used to describe different inference rules which extend Transfer Semantics so that more complex queries may be handled. Six inference rules are introduced. This chapter shows how a rule called the Rule of Composition can act as a plan generator.

- Chapter 5 describes how the Rule of Composition may be extended to allow specific types of composition. Some new rules are created to compute selective, parallel, and forked composition. There is a need to represent parallel actions, as many operating systems such as UNIX allow parallel executions of commands. Moreover, people can ask natural language queries about such parallel executions.
- Chapter 6 shows how natural language input may be understood by parsing queries into meaning representations. A theory of embedded action representations is introduced and examples of meaning representations for various queries are described.
- Chapter 7 contains a discussion on the methodology and architecture of the artificial communicator called OSCON (Operating System CONsultant). A program called PlanCon, which computes rules of inference over knowledge structures, is described. PlanCon is deployed in understanding complex plans appearing implicitly in user queries.
- Chapter 8 is a summary of work reported in this thesis. We discuss the value of this work, through a critical analysis. As always, future directions of research are predicted.

Chapter 2: Designing consultant systems

This thesis will describe a design for an operating system consultant that will accept English queries and answer the queries in English. There are different approaches to building operating system consultants and we will describe each in turn with their advantages and disadvantages. Some of the approaches will be very close to our own, while others could not be further away. Some systems will have been designed for specific operating systems while others may be general.

2.1. Evaluation by criteria

In evaluating any consultant system we should define some criteria that will be useful. We define four criteria for evaluating consultant systems: (1) friendliness of the interface, (2) detailed answer production, (3) system architecture, and (4) reference to other systems.

By **friendliness** we mean how usable the system is. Simple on-line help systems like the UNIX *man* program will work only if the user knows command names or some keywords for what he wants to do. However, to obtain help on various concepts the user must know the command, or related keywords, and that isn't friendly at all. Any good consultant system should answer queries by concepts, not only keywords.

Simple help systems will not answer complex queries. Certain processes, such as printing screen output on the printer, require the concatenation of several commands with multiple options. A simple help system like UNIX man package would force the user to retrieve information on several commands, sift elements from each, and work out the correct format for the complete process. Consultant systems should fuse information together for detailed processes and they should supply **detailed answers**.

Further learning of any system by the user is accelerated if he formulates a good model of the system in his mind. Such system models are called Cognitive models by psychologists. Cognitive models are necessary if a user is to properly understand how to use the system and for the formulation of further intelligent queries. A user who has not used UNIX before may come across the concept of *pipe* and need to know what it means. Someone may like to know the whole structure of the UNIX file system. Many help systems do not provide information about such **system architecture**.

Any consultant system must include **reference to other systems**. New users may have a perfect background in the use of certain computer operating systems and may ask queries in terms of these. If related terminology is not built into the consultant, it will fail. In fact, a user may understand operating system concepts very well but not know that he is using the wrong terminology.

2.2. Types of operating system consultant

Many researchers are working on building operating system consultants. There are four basic types of system available: (1) simple keyword based systems (SIMPLE), (2) systems with menu-based interfaces (MENU), (3) systems with limited natural language capability (LNL), and (4) systems with natural language interfaces (NL). Although we will discuss the characteristics and examples of each type of system, we are not going to describe all of today's systems. There are many of them.

2.3. Simple keyword systems

Examples of SIMPLE systems include the UNIX *man* and *apropos* facilities. The problem with these facilities is that they are very unfriendly; the user must know the name of a keyword before he can find out some information about one. Also, these systems do not provide detailed information when a user tries to find out about linking a number of commands together. The user does all the work himself. SIMPLE systems do not usually include information about other systems. Therefore, SIMPLE systems do not perform well under the four criteria of evaluation.

2.4. Menu-based systems

MENU systems are based on menu-selection where the user is presented with a menu displaying a number of options. The problem with most menu-selection approaches to operating system consultation (pointed out by Wilensky et al. 1984, p. 576; Hegner 1987, p. 1; and McDonald & Schvaneveldt 1987, p. 14) is that they are not very useful if a user knows what he wants to do, but does not know the explicit command for doing it. Such menu systems usually key on the names of commands. Therefore they behave badly in terms of friendliness. To find help about some concept a person must know the name of the appropriate command or a related term to do so. However, it is usually the name of the command that the user requires in the first place.

McDonald et al. (1983) have organized studies to clarify the effects of menu organization on user performance. They used explicit targets (e.g., ‘‘lemon’’) and single-line definitions (e.g., ‘‘a small, oblong, pale-yellow citrus fruit’’) to examine the effects that *type* of target has on menu-selection performance. They point out that real-world users seldom search for explicit targets in menus. If people know exactly what they are looking for, then they probably know where to find it. Say a user is looking for some command to remove a file. It is unlikely that the name of the command is known. Searching the menu system is easy if the user knows that the command is delete. But, then the user need not use the menu system at all.

It is possible that one could build a menu system where abstractions or concepts such as printing are represented. However, such abstractions may still not be useful to some user who can describe what he wants to do, but cannot find any mention of that in the set of abstractions. The problem arises because natural language expressions are at such an abstract level that they may not fall into any set of concepts. You may argue that such abstractions can also be built into a MENU interface. That is true, but then what you have is a natural language front end. Natural language front ends are closely related to MENU systems which contain many abstractions. Such front ends allow users to specify queries in terms of abstractions of word meanings, and are therefore more flexible. It is important to point out that we are not saying there is anything wrong with menu-selection approaches. They are a useful insight into how to structure knowledge about some domain and are a useful first draft at building any interface.

MENU systems behave badly with complex queries involving a large number of constraints. For example, say a user wants to delete a listing of a file on the printer queue. The menu system will not present such complex information, and the user must piece together knowledge about the many commands involved in this process.

Any user needs to have a good education on the structure of a system. There needs to be some mechanism for reporting the existing structures in the system and how they are related together. Such static aspects of the system are not associated with any particular command but to each and every one of them. MENU systems perform very well under this task as it is easy to show detailed pictures on the screen. Also by using the menu structure many times the user obtains a good cognitive model on the structure of the system.

MENU systems can also behave very well under the consideration of reference to other systems. MENUs easily provide options that allow the user to see each command as it would appear on other systems. The next four sections involve descriptions of MENU systems.

2.4.1. McDonald & Schvaneveldt

The Cognitive Systems Group at the Computing Research Laboratory are developing formal methods for interface design (see McDonald et al., 1986; McDonald & Schvaneveldt, 1987). McDonald and Schvaneveldt have defined theoretical motivations for their empirically based approach along with a related discussion of scaling and knowledge acquisition techniques. One application to illustrate key aspects of their methodology is an ongoing investigation of UNIX users aimed at improving on-line documentation systems. They are developing a theory of structural descriptions for UNIX. These will be useful in building a menu-based consultation program which will allow users to efficiently develop accurate conceptual models of operating systems.

Their UNIX interactive documentation guide (Superman II) (1) is based on empirically derived representations of experienced users' conceptual models, (2) has several perspectives (e.g., functional and procedural), (3) has multiple levels of abstraction within each perspective, and (4) provides users who are familiar with other operating systems (e.g., DOS∗) a 'bridge' for transferring their knowledge to UNIX. Their knowledge representation is based on proximities of semantic information and is organized in a network structure. The networks are described in

∗ DOS is a trademark of International Business Machines Corporation.

Dearholt et. al (1985). We are working closely with the Cognitive Systems Group to provide empirical backing for any assumptions made in developing the natural language understander.

2.4.2. Hayes & Szekely

Another menu-selection approach is described in Hayes (1982) and Hayes & Szekely (1983). They have designed a system called COUSIN which is a *command-level* interface for operating systems. By command-level we mean the system will execute commands from the interface. The COUSIN system provides two types of user friendly information: (1) static descriptions of possibly invoked subsystems, including their parameters and syntax, and (2) dynamically produced descriptions of the state of current interaction. One of the applications of the COUSIN interface is to provide a command-level interface to the UNIX operating system, i.e., to provide an alternative to the standard UNIX shell. COUSIN consists of a network of text frames connected by named semantic links. Each frame is variable in size and contains less than a screenfull of information. COUSIN shows to the user information that is hidden from him by a natural language understander. While using a natural language understander the user does not see, or need to know, the structure of stored knowledge. However, such information can be found by asking the right questions. The knowledge exists and is presented to the user only on demand. The natural language understander informs the user, in terms of English, the specific pieces of stored knowledge that are particularly relevant to some query.

2.4.3. Tyler & Treu

Tyler and Treu (1986) describe an adaptive interface design, and a prototype user-computer interface, to demonstrate both the feasibility and utility of a general adaptive architecture. The system is a command-level interface where the interface takes a user's entry and sends a valid command to the operating system. A prototype has been designed which will interface the user to a UNIX operating system. Features of the interface are geared towards the particular user, and the specific task currently being executed. The prototype does not provide all possible UNIX commands, but it does make the most commonly used commands accessible through the interface. There are a number of textual pieces of information which can be used in giving help on some command. However there is no great theory of how to efficiently represent knowledge about UNIX here.

2.4.4. Billmers & Garifo

Billmers and Garifo (1985) are building knowledge-based operating system consultants. They have implemented an expert system called TEACHVMS which is used for helping TOPS-20∗ users learn about the VAX/VMS∗ operating system. They are also developing a system called TVX which provides a general operating system shell useful for designing specific operating system consultants. Both of these systems are menu-based expert systems. In agreement with our approach, Billmers and Garifo are interested in planning solutions to complex user tasks, requiring many steps. The fact that TEACHVMS converts TOPS-20 commands to VMS commands means that it must contain similarities between concepts from different operating systems. This concurs with our criterion of reference to other systems. TVX contains knowledge in two forms: abstract operating system concepts, and knowledge specific to the target system (i.e., VMS).

∗ TOPS-20 and VAX/VMS are trademarks of the Digital Equipment Corporation.

2.5. Limited natural language systems

LNL systems behave like expert systems where the user asks queries using limited natural language. However, the natural language capability of LNL systems is not extensive enough to allow adequate concept formation for complex queries. LNL systems are more friendly than SIMPLE or MENU systems as they allow more flexibility in the input. LNL systems allow detailed constraint input and can be good at describing the static structure of the system. LNL's may also include reference to other systems in their knowledge representation.

2.5.1. Yun & Loeb

The program CMS-HELP developed by Yun and Loeb (1984) is an example of an expert system. CMS-HELP serves as an on-line consultant for users of the VM/CMS∗ operating system. The system assists novice or experienced users who need to use unfamiliar system facilities. Advice is given in terms of the sequence of commands needed to accomplish some user task. The CMS-HELP expert system was constructed using EMYCIN, a program for developing knowledge-based consultation systems.

2.6. Natural language consultant systems

The advantages of natural language understanders over most LNL and MENU approaches are numerous. We will not discuss those advantages here, as this has been done elsewhere (see Douglass & Hegner 1982, p. 1; Wilensky et al. 1984, p. 576). These types of system perform very well under each criterion of evaluation. The input is in terms of natural language and therefore there is a very large varied input. NL's are very friendly and they supply detailed answers if the knowledge

∗ VM/CMS is a trademark of International Business Machines Corporation.

representation is detailed too. Reference to other systems can be encoded into a knowledge representation. The disadvantage of NL systems is that are very difficult to build.

2.6.1. Hegner & Douglass

Hegner and Douglass developed a natural language UNIX help system called UCC (see Douglass & Hegner, 1982 and Hegner & Douglass, 1984). UCC was a prototype system, implemented in Franz Lisp on a VAX-11/780. It used a simple natural language front end based on augmented transition networks. The output of the parser filled slots in so-called *case frames*, which represented the structure of common queries. The development of a knowledge base and query solver were not advanced enough so that they could be linked with the front end. Therefore, UCC generated answers to queries directly from concept case frames rather than from any particular formal language. The system was tested at Los Alamos National Laboratories and it could answer a surprising number of queries adequately. Also, the information obtained with test runs was useful in identifying its shortcomings.

There were two major problems with UCC: (1) As the front end included a relatively simple knowledge base, it was unable to answer sophisticated queries with many constraints involving command options. It could tell the user that the *chmod* command is appropriate for changing file protection, yet it was unable to give specific directions for changing the protection to a particular mode. This could be rectified by linking in the more detailed knowledge base. However, another problem was more serious. (2) The simple augmented transition network method of parsing was not sufficient enough to handle the types of queries posed by many users. It became very apparent while improving the system that the best approach would be to develop a new, more sophisticated design for the natural language front end.

A program called Yucca was an attempt to augment the UCC system in two ways. Yucca incorporated a much more sophisticated formal knowledge base and an improved natural language front end. The knowledge base was implemented at Los Alamos and the University of Vermont. However, due to funding restrictions this work halted and the system was only partially implemented.

2.6.2. Wilensky et al.

At Berkeley, Robert Wilensky heads a group who have built an understanding system called Unix Consultant (UC) which processes natural language queries about UNIX (see Wilensky, 1982; Wilensky et al. 1984, 1986; Wilensky, 1987). Our approach to consultation is similar and yet different to the one at Berkeley. We are both building natural language systems, yet the way we do that is quite distinct. In UC there is no separation and formalization of detailed knowledge on operating systems in a knowledge base. All aspects of UC make use of one general knowledge representation called KODIAK (see Wilensky, 1986). This compares to our approach of having abstract knowledge in the natural language understander and detailed knowledge in a knowledge base. Another distinction is that presently the UC program is intended to be an operating system consultant for UNIX, whereas our system is intended to maintain references to other systems. In building the natural language understander we are particularly concerned with understanding complex queries where there are a number of operating system commands interrelated with each other, to denote higher level processes.

2.6.3. Kemke

Kemke (1986, 1987) describes an intelligent help system called the SINIX∗

∗ SINIX is a UNIX derivative developed by SIEMENS AG.

Consultant (SC) for the SINIX operating system. The system is intended to answer natural language questions about SINIX concepts and commands. SC has a rich knowledge base which reflects the technical aspects of the domain as well as the users view of the system. Although SC incorporates a knowledge base which contains similar knowledge as our natural language understander, there is no separation out of the detailed knowledge needed to answer or solve user queries. Therefore, we see SC as being similar in approach and design to the UC system.

Chapter 3: A knowledge representation

Any good consultant system must include a knowledge representation of the domain upon which it is meant to consult. A good theory of how to represent that knowledge is necessary if the consultant is to be efficient and useful. This chapter is about defining what we think is a good knowledge representation for operating systems.

We need to investigate the means by which various operating system concepts can be formalized using an appropriate knowledge representation mechanism. This representation can then be used effectively to understand natural language expressions involving different concepts. We assume the philosophy of Wilks (1978b, p. 210), "The emphasis here is the reverse of the conventional one in this field: we stress the form of representation of language and seek to accommodate the representation of knowledge to that, rather than the reverse."

3.1. The elements of a representation

It is our belief that people think (however abstractly) of operating system commands in terms of inputs and outputs. People see commands as sets of states of objects before and after a command is executed. Each command is a black box which takes a set of objects as input and produces another set of objects as output.

People ask questions about operating systems in the same way that they think about commands. It turns out that most English queries about operating systems involve users expressing the goal of obtaining some command. Commonly, users will try to describe the affect of a required command on some object(s). For example, in the query, "How do I print out a file with pagenumbers?", the user is expressing the need for a command to print the object *file* with the object *pagenumbers*.

It is the constraints specified in a user query that enable us to recognize a command. Therefore, it seems useful to build knowledge structures for describing commands so that these structures are closely related to possible natural language expressions of such commands. Natural language queries involving descriptions of commands can be parsed into some high-level meaning representation. To interpret queries effectively we need access to domain-specific knowledge. Such knowledge could be formulated as abstract representations of actions or objects which are matched to natural language representations in order to decipher them.

3.2. A representation for objects

There needs to be some data structure for representing operating system objects. Frames have been used before in AI (see Minsky, 1975) for knowledge representation. We can use frames to contain information about various objects in the system. If objects are not linked together in some way, then they will have little meaning in the system. Another useful tool is the ability to specify relations between different objects. Relations within knowledge representations have been implemented before using hierarchies of objects. Examples of hierarchical representations are found in Bobrow & Winograd (1977), Brachman (1979), Fass (1986a, 1986b) and Goldstein & Roberts (1977).

Various operating system objects such as *files*, *protection*, *command-name*, *last-read-time*, *creation-time*, and *password* can be represented by object frames. Object frames should exist statically in the system before any processing begins. Each object frame should contain two types of information: (1) the information specific to some object, and (2) the relation between an object and others in the system.

We call the former *nodes* while the latter are called *arcs*.

Each node is a set of attributes characterizing an object frame. Nodes in object frames could be specified using an identifier like *has*. It is possible that *has* relations will contain other object frames.

3.3. A hierarchy of objects

In any hierarchy of objects which are linked together there may be many ways of defining relations between them. First of all there should be a link to specify one object as being a type of another. This is useful because we notice that files are types of container and directory files are types of file. We can call such a relation, a *type-of* relation.

Also, certain objects are not types of others but parts of them. For example, protection is part of a file and so is user id. *Creator* and *last-tape-read-time* are also parts of files. These can be called *part-of* relations.

Certain commands are instances of others. So, the commands *lpr*, *cat*, *cp*, and *pr* are all related to the *command-name* object frame by an *instance-of* arc relation.

In Figure 3.1 below there is an example of what the object frame for protection-type should look like.

Protection-type is a part of protection and includes user designators, access privileges and file access. The object frame for user-designator is shown in Figure 3.2.

It is noted that each user-designator is a type-of designator. User-g, user-u, user-o are all instances of user-designators (see Figure 3.3).

In the node set for protection-type (Figure 3.1) the second *has* relation specifies an object frame called access-privilege. *Read*, *write* and *execute* are all instances of

```
(o-frame protection-type
    (arcs (part-of protection))
    (node (has user-designator)
          (has access-privilege)
          (has file-access)))
```

Figure 3.1. Object frame for protection type.

```
(o-frame user-designator
    (arcs (type-of designator))
    (node ()))
```

Figure 3.2. Object frame for user designator.

access-privilege. Finally, *file-access*, contained in the third *has* relation for protection-type, has instances *access* and *no-access*. It is already apparent that objects need to be related in a complex hierarchy. In Figure 3.4 we show a description of some of the hierarchy.

From Figure 3.4 we have the relations in Figure 3.5. Figure 3.5 shows *directory-password* (a concept from the TOPS-20 operating system) which is defined in terms of UNIX concepts. This will be particularly useful for helping some user who is confused as to which operating system he/she is using. In fact it is one of the

```
(o-frame user-g
      (arcs (instance-of user-designator))
      (node ()))

(o-frame user-u
      (arcs (instance-of user-designator))
      (node ()))

(o-frame user-o
      (arcs (instance-of user-designator))
      (node ()))
```

Figure 3.3. Instances of user-designator.

criteria specified in Section 2.1 that any good consultant system should contain such referents between different operating systems. In the hierarchy in Figure 3.6 we show the definition of a file in terms of its components.

3.4. The representation of commands

We know that commands are actions which define transfer relations between objects. Knowledge structures for commands are necessary in the system. A good way to represent actions is to determine the existence or states of objects before an action occurs and also the states after the action has executed. Let’s call the states before, *preconditions* and the states after, *postconditions*. There should also be some mention of the person who can perform a particular action. A representation for commands should include preconditions, postconditions, and an actor. The question that arises next is, “What amount of information should be represented?”

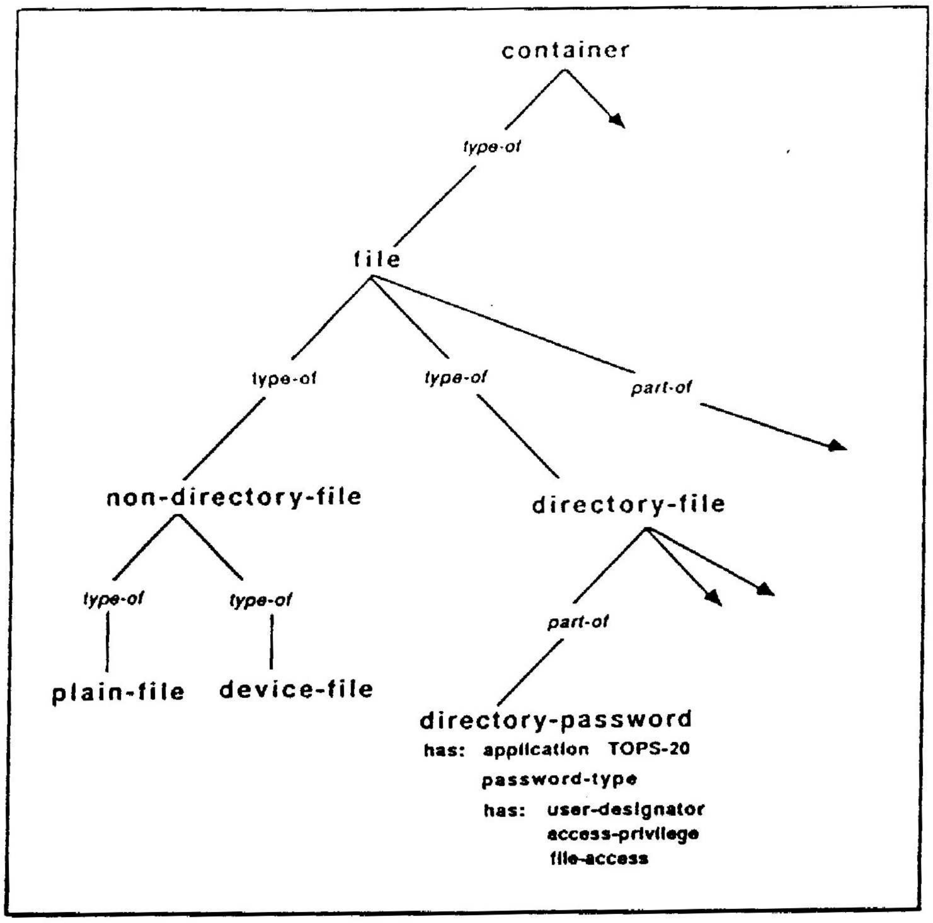

Figure 3.4. Hierarchy of file objects.

We could represent the preconditions and postconditions for every command that a user could ask about. However, that would take a very long time to do, and it would take a long time to search the database of actions to find the right one, because you see, every command would be represented as an action. There seems to be no way out.

```
plain-file
  is a type of non-directory file
    is a type of file
      is a type of container


directory-password
     has application TOPS-20
     has password-type
       has user-designator
       has access-privileges
       has file-access
```

Figure 3.5. Relations from object hierarchy for file.

Yet, “What do we notice about certain commands?” They can be grouped together under certain categories. For example, the commands *lpr*, *pr*, and *more* all involve printing information from a file although they do that in different ways. Also *rmdir* and *rm* involve removing objects. Already, we can see a solution to a massive searching problem. We can represent abstractions of commands as actions. There will be an action representation for *printing* and one for *mailing* and one for *deleting*, *listing*, *moving* and so on. Have we lost anything by abstraction? Yes. Even if we have an action representation which has preconditions, postconditions and actors we have lost the names of the various commands that perform particular cases of an action. Yet, all we need to do is place the commands in the representation itself and we call the whole structure an action frame. We call preconditions, postconditions, actions, and actors, frame components.

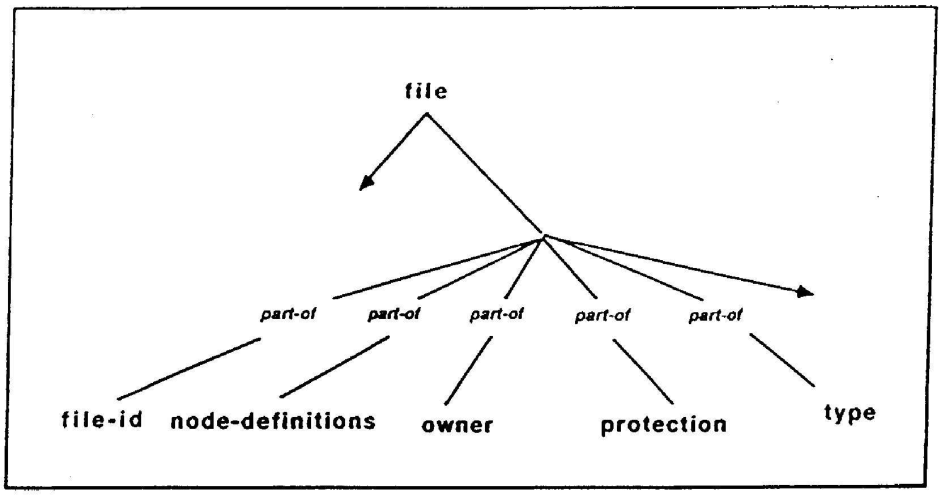

Figure 3.6. Hierarchy of objects to define parts of a file.

3.5. The necessity of preference

The next problem to be solved is what sort of information should be placed in each frame component. Well, the preconditions and postconditions should contain information about objects and how the various commands can effect them. We cannot define all the objects affected by some action. That would take up too much space. A scheme must be defined where only *preferred* objects are represented. By preferred objects we mean objects that are usually affected by some action. Not only must objects be represented in each condition set but the relations between objects should be there too. Such relations will also be preferred as there may be infinitely many relations for a given action.

This idea of preference is not new in AI. It has been used before by Wilks (1975a, 1975b, 1978b) in Preference Semantics and Fass (1986a, 1986b) in Collative Semantics to formulate correct interpretations of natural language sentences. A good discussion on the relative merits of various types of preference are found in Fass & Wilks (1983). In Wilensky (1987) there is a description of *concerns* which are preconditions particularly relevant to a given plan. The term *concern* is synonymous with our concept of preferred conditions.

3.6. A dilemma in “weak” versus “strong”

Another question arises as to what level of detail each object should be represented. We could represent objects at their most detailed level or they could be represented at their most general or abstract level. Plain files, non-directory files, and device files are all types of file. Therefore, a file could be represented as just *file* or *plain-file* or *non-directory-file* or *device-file*.

We must consider the disadvantages and advantages of choosing specific or general representations. To do that we consider what the representations are being used for. The representations are being used to understand user queries.

3.6.1. Preconditions

People tend to talk about printing files rather than printing plain files or device files. As people tend to specify weak preconditions, and as we want to match these knowledge structures to what people say, then we should use weak preconditions in the action frames too. Therefore, we try to make the preconditions of an action as weak as possible. Notice that we use the word *tend* here, i.e., people can mention strong preconditions in their queries although that is not what they *usually* do. Already we have specified a preference to having weak preconditions for action frames. Therefore we represent files at a general level as *file* in the precondition set.

It is important that we allow some mechanism whereby stronger preconditions can be derived if they are needed. It wouldn't do for the action frame precondition sets to know only about files and not plain files or directory files or any other types of files. "What do we do?" The object hierarchy, described earlier, has information about files and their types. Therefore it is possible for the system to derive more specific objects if they are needed. The power of action frames lies here. Even though action frames are abstract, more specific frames can be generated easily by inferencing on the object hierarchy and inserting new objects in the action frames. We will show how that can be done in the next chapter.

A question of efficiency arises as preconditions were defined to be as weak as possible. Say we were to represent strong preconditions. Then many of them would have to be represented. There would be a lot of precondition objects and the only way to minimize these would be to represent at best a minimal set. Therefore it was a good move to represent more general objects, just like we represented more general actions, and we can derive the more specific objects from the object hierarchy if they are needed.

3.6.2. Postconditions

Postconditions for any action are changes in object states resulting from the execution of that action. In all action frames the postconditions represent changes in state of the precondition set. People tend to specify strong postconditions when asking queries. For example, someone may say, "How do I print a file with pageheaders and line numbers?". People specify such strong postconditions as *pageheaders* and *line numbers* because they want to define precisely what an action or command should do. Therefore we have a second preference for strong postconditions as people use them in queries. In the postcondition set we include objects such as *non-directory-files* rather than *files*.

We try to represent the “strongest” postcondition set for any action. By strongest we mean the maximum number of (or most constraining) postconditions necessary to characterize some action sufficiently. We know *non-directory-files* to be types of file (Figure 3.5) and that either could denote postconditions for printing files. However, the use of non-directory-files (strong) as a postcondition for printing rather than files (weaker) is a more precise definition about the effects of printing. That is why we reflect non-directory-file in the postcondition set rather than file. There is no harm in weakening the postcondition set when that needs to be done.

One could argue that we are not being efficient in representing postconditions as they involve strong rather than weak information. There will be many of them. However we claim that strong postconditions, even though represented less efficiently, will make the system more globally efficient. That happens because there will be much time saved in not having to infer strong postconditions from weaker ones (if we had used them) many times. And, ‘‘Why would we have to do that many times?’’ Because, people usually mention strong postconditions when they ask queries about operating systems.

3.7. The representation of conditions

Conditions on actions can be represented as constraints on objects. Objects and relations between them will be preferred. ‘‘How do we know what preferred objects and relations are?’’. That is an empirical question and will not be discussed further here. However, we will assume that certain objects and relations will be usual for some action. For any frame there will be certain preconditions that should not exist. For example the precondition set for printing should specify the non-existence of directory files because people do not usually print directory files. These types of conditions are called *mandatory*.

Mandatory conditions are useful for a very important reason. Say mandatory conditions did not exist and files (weak) were represented in the precondition set. As directory files are types of file (Figure 3.4) the system can now infer (wrongly) for the PRINT∗ frame that directory files are printable. Yet, this will not happen because by using mandatory conditions which override all other frame conditions the problem disappears. The mandatory condition for print declares directory files to be non-printable.

It seems likely that the postcondition sets will not always contain mandatory postconditions because these will have been recognized by the precondition set. Therefore, mandatory postconditions will only be concerned with problems where specified output can not be obtained from correctly specified input. We would only enter those mandatory postconditions which people usually get wrong.

There should be a set of conditions which are *optional* for some action. These would involve file contents being visible-byte-sequences and the existence of printers. Also, we define *default* conditions for each action frame which contain the most general information about an action. For example, for the print frame the most general information about printing would be that one prints files on the screen and that would be the default.

3.8. Linking preconditions to postconditions

It would be useful if we could define a correspondence between preconditions and postconditions. Such knowledge would aid in predicting the most likely postcondition (or precondition) for some explicitly mentioned precondition (or

∗ We use upper case letters to denote any action frame, or information contained in one.

postcondition) in a user query. These predictions would help in understanding user queries, because it would be a useful check if predicted information matched that coming in. It is the ability to predict preconditions and postconditions for user queries that will give added power to the system. Otherwise, it will be difficult to formulate a good meaning structure to be solved by the knowledge base.

A correspondence is expressed implicitly between the optional conditions in the precondition and postcondition sets. Say, $\{ P_0, P_{1,} \ldots P_n \}$ denote the optional preconditions for some action A. Then, these are related to the optional postconditions $\{ Q_0, Q_{1,} \ldots Q_n \}$ so that P_0 <=> Q_0, P_1 <=>* $Q_{1,}$ ... P_n <=> Q_n for action A. So, the first optional precondition in the precondition set corresponds to the first in the postcondition set, the second to the second, and so on. One-to-one correspondence between preconditions and postconditions is implicit: it is the *position* of a particular condition in its precondition/postcondition set that determines correspondence and there are no other markers to specify that.

As we represent postconditions at a stronger level than preconditions, many situations will arise where a given precondition will correspond to, or map onto, two or more postconditions. If we decide to write out all the optional preconditions explicitly then there will be some redundancy in the precondition sets as some may be represented twice.

However, we shall write out preconditions and postconditions as there should not be too many of them for a given frame. If it turns out that many conditions need to be repeated exhaustively for action frames, the correspondence can be denoted explicitly by some flagging system rather than writing the same preconditions twice. The fact is that the cost of a checking algorithm for flags may be too costly if there are not too many conditions for most frames.

* “<=>” denotes *corresponds to*.

3.9. Actions and actors

It is also necessary to specify the possible actions that cause transfer between preconditions and postconditions. Associated with each action will be a number of options. Actions will include commands like *lpr*, *lpq*, and options include -l, -a, -s and so on. We call the complete organization of object frames, object hierarchy and action frames, *Transfer Semantics*.

3.10. The correspondence between queries and knowledge

In this section we will show how various queries could be interpreted using object and action frames. While doing this we keep in mind that action frames are representations for describing operating system commands.

We adopt a distinction between concept description queries and dynamic queries. This distinction has been emphasized by Hegner (1987). Concept description queries are simple queries about objects which involve no manipulation of those objects. Typical concept description queries are, “What is a directory?”, “What is pr?”. Dynamic queries are those which involve actions transferring objects. Typical examples are, “How do I print a file on the Imagen?”, “What is the option on the cat command which numbers lines?”, “How do I print a listing of my directory?”.

3.10.1. Concept description queries and knowledge

In handling concept description queries such as, “What is read protection?” the hierarchy of object frames becomes very useful. From the shapshot of the network in Figure 3.7 it is possible to locate relevant object frame relations. The following section of network is used in generating a static domain-specific representation of the latter query.

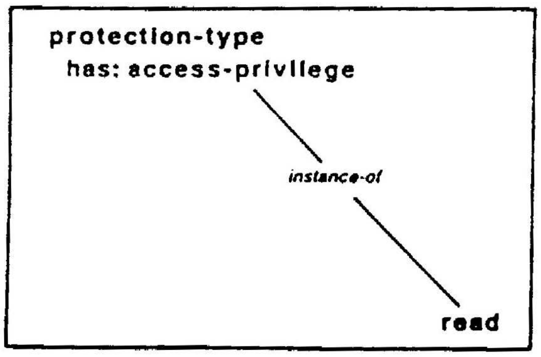

Figure 3.7. Snapshot from object hierarchy for protection type.

Now, say some user has used the TOPS-20 operating system for most of his computer lifetime and decides to use UNIX for a change. Then he/she is likely to assume that UNIX is similar to TOPS-20. One could expect queries such as, “What is the permanent storage limit?”. The relations in Figure 3.8 are used here.

```
directory-file
    has permanent-storage-limit
          has application TOPS-20
          has similarity disk-space-hard-limit
```

Figure 3.8. Similarities between different operating systems

The above relations denote the similarity between concepts from two operating systems. The similarity between disk-space-hard-limit and permanent-storage-limit is marked using has relations. This mechanism is especially useful if a user thinks in terms of one operating system but is using another.

3.10.2. Dynamic queries and knowledge

The object hierarchy is availed of again for dynamic queries. However, as dynamic queries involve actions, action frames must be referenced. Say, for example we want to interpret the query, “What is the option on the cat command which numbers lines?”. Through searching the action frame preconditions, a precondition mentioning files and their containing visible-byte-sequences would be matched. The relevant postcondition is specified by the file again appearing on the screen and having a line numbers filter applied. This is done by moving down the object hierarchy from *filter*, which occurs in the postcondition, to *numbered-lines*, which are a type of filter. Also, the *cat* action in is marked because “cat” was mentioned in the query. The user is marked as being the relevant actor.

Similarly, the query, “How do I print a file on the Imagen?” matches an optional precondition where files contain visible-byte-sequences. Also, the existence of a printer queue is needed. The postcondition will specify output coming out on a printer rather than the screen. There will be no match for the action component of a frame, as no action was mentioned, and the actor is again *user*.

3.11. Alternative representations of knowledge

The UNIX Consultant (UC) (see Wilensky et al. 1984, 1986) emboies a knowledge representation called KODIAK. The central theme of KODIAK is that it is a relation-based system. Wilensky (1986, p. 23) says, “... KODIAK relations have a fixed number of argument positions. Moreover, each argument position of a relation is itself a full-fledged object. In general, the meaning of these argument-objects is derived from the named relation that hold between them." KODIAK has a wide representational scope and still maintains the possibility of conforming to a canonical form. At the action frame level Transfer Semantics is also a relation-based system where actions are described in terms of precondition-postcondition correspondence. In Transfer Semantics the meaning of any action is the precondition and postcondition set for that action. Wilensky decides to represent all concepts in terms of relations. We only see the need to represent actions (which manipulate objects) with relations. Many objects are not defined by relations in Transfer Semantics although there may be relations between them.

The UC system is not presently intended to handle queries using terminology from operating systems other than UNIX. We are more concerned with understanding complex queries where there are a number of operating system commands interrelated with each other, to denote some higher level process. It seems that Transfer Semantics, which captures the meaning of commands, in a way that people do, is a suitable formalism for abstracting operating system behavior.

Kemke (1987) has independently come up with a knowledge representation for operating systems quite similar to our own. The SINIX Consultant Knowledge Base consists of a taxonomical hierarchy of concepts according to different views or classifications of domain concepts. These domain concepts are commands and virtual objects from the SINIX operating system. Higher level concepts correspond to natural language terms, mental model entities, or more general abstract actions and objects. Each concept is described with respect to its function, structure, use, and/or relation to other concepts. The SINIX knowledge base is organized, like ours, as a hierarchy of concepts. The leaves of the hierarchy correspond to SINIX objects or commands. Kemke also includes preconditions and postconditions in her definitions.

There are some differences with our approach. We do not include actions in a hierarchy, although action frames do make use of the hierarchy. In Kemke's knowledge representation actions are stored explicitly at different levels of detail in a hierarchy. For example, *communicate-with-user* is stored at a level above *send-mail* and *read-mail* in the hierarchy. However, these are not stored as different actions in a Transfer Semantics hierarchy. We represent actions abstractly, and as frame objects are defined in the hierarchy it is possible to **compute** new action frames (which are stronger or weaker) from these object definitions.

The work by Schank (1975) on Conceptual Dependency (CD) is largely concerned with representations for actions. He proposes that complex representations are composed out of a well-defined set of primitive concepts. The central theme of CD is that meaning representations have canonical form so that things meaning the same are represented in the same way even though they may be expressed differently. CD involves decomposition into primitives where all complicated entities are represented by simple elements.

The problem with CD is that too much concentration on canonical form has led to a lack in formalizing any specific high-level objects. Wilensky (1986) shows that CD poses problems in computing certain types of inference. It is the very decomposition into primitives that causes inference problems. For example the system tries to make inferences about *buying* in terms of the ATRANS (abstract transfer) primitive. As Wilensky (1986, p. 12) puts it, "...decomposing conceptual objects into primitives doesn't help one make inferences any more than it gets in the way. It facilitates inferences about more abstract ideas like for example, change of possession, only at the cost of making it more difficult to make inferences about more complex ideas such as buying."

Transfer Semantics is similar to CD in that it embodys representations for actions. However, the difference is that TS is a mechanism for describing the more complex effects of performing specific actions on particular objects. It is important to represent the complex differences between actions as well as their similarity and this is exactly where CD fails. Certainly, CD would present many problems if we were to use it in any formalization of operating system actions or commands. In fact, Wilensky et al. (1984) and Arens (1986) found this out when they used CD in an earlier version of the Unix Consultant (UC).

Our object frames are similar to the frames proposed by Minsky (1975). Yet, Minsky (1975, p. 234) decides, "that any event, action change, flow of material or information can be represented by a two-frame generalized event." This is in contrast to our system where single action frames are used to represent state changes of objects. Wilks (1978b) describes semantic structures called pseudo-texts for natural language understanding. Wilks (1978b, p. 203) defines a pseudo-text as "...a structure of factual and functional information about a concept or item, and is intended to fall broadly within the notion of frame in the sense of Minsky, Charniak, and Schank..." Pseudo-texts are also similar in function to the object frames we describe herein. Our action frames have similarities with the scripts discussed by Schank & Abelson (1977). Action frames could be interpreted as scripts representing the behavior of various operating system commands.

3.12. The limits of knowledge

It is concluded that Transfer Semantics is an appropriate mechanism for describing actions and how these actions transfer objects. It seems a particularly effective mechanism for abstracting characteristics of various computer operating system actions in a concise formalism. We have shown how Transfer Semantics could be used to specify domain-specific knowledge in order to interpret concept description and dynamic English queries. The use of Transfer Semantics in an operating system consultant will enable the production of detailed representations of user queries. These representations will represent how the system understood a natural language query.

A particularly useful feature of Transfer Semantics is that similarities between object frames are marked. Therefore, even though a query may be presented to the understander with TOPS-20 lingo, that query can be interpreted and answered in terms of UNIX. It is hoped that Transfer Semantics will be used to model other operating systems as research continues.

In building our knowledge representation for operating systems there have been a few things left out. There was no attempt to describe how OSCON gets the right frame for some query. Such processes will be discussed in Chapter 6. Nor, have we described the meaning representation of an English query before the frames are matched to it. These representations are also discussed in Chapter 6. However, even more important is the fact that the knowledge representation itself may not be complete.

Sets of conditions for action frames are only preferences in the system that are typical of some action. We use preferences for two major reasons: (1) in order to select the correct frame, and (2) if we specified all possible transfer conditions on frames they would certainly become very large. Yet, the action frames are not restricted to preferred conditions because of their relation with the object hierarchy.

The query, ‘‘How do I print a plain file?’’ cannot be handled by Transfer Semantics as it stands. That is because preconditions are represented weakly and plain files do not occur in the PRINT frame precondition set. We know the object hierarchy can be used to derive a more specific action frame to handle this query. However, we must work out exactly how that is done. Questions arise as to how we know when to use the object hierarchy to find new information about objects in the frames. More important, what sort of rules define how we do this. Some queries may involve more than one action and there must be some rule for adding actions together. The solutions to these questions are tackled in the next chapter.

Chapter 4: Some rules of inference

In the previous chapter we discussed and developed a knowledge representation for operating systems called Transfer Semantics. The job of Transfer Semantics is to act as a good knowledge representation which can be used in understanding the queries people ask about operating systems. There were certain elements of Transfer Semantics that were not explained in depth. For example, ‘‘How does the system know when to use the object hierarchy to obtain new knowledge?’’ Moreover, ‘‘What does OSCON do when it finds out that new knowledge is necessary?’’ ‘‘What happens if someone uses more than one command in a user query?’’ It is important to write out explicitly the mechanisms to solve these questions. That is what this chapter is about.

4.1. A need for inference rules

A knowledge representation scheme is never complete while there are no strategies to manipulate that scheme. As conditions on action frames are preferred, we choose those conditions typical for some action. This is done for three reasons: (1) so that the correct frame will be selected for a particular query, (2) frames would become very large if all possible transfer conditions were specified, and (3) inherent requirements for specifying *weak* preconditions and *strong* postconditions on frames. The very fact that frames contain only preferred conditions means that Transfer Semantics is weak. That can be shown explicitly by various examples. The power of Transfer Semantics must be increased.

Some interesting problems arise when Transfer Semantics is used to understand natural language queries. A clue to such problems was already given above.

Only *preferred* conditions on frames are deployed. Otherwise, the frames would become enormous and difficult to handle. Also we showed in Chapter 3 why it would be beneficial to represent weak preconditions and strong postconditions in frames. However, by doing that we have constrained the preconditions and postconditions and that causes problems. Let's look at some of the problems.

Say some user decides to enter the query, ''How do I print a file on the screen?'' This query will be parsed first, into a shallow form, and then into a semantically deeper meaning representation. So far there is no problem. The next step in the control flow of the understander would be to select a domain-specific action frame. The PRINT frame should be selected. However, that may not happen as the postcondition set for the PRINT frame only knows about specific NON-DIRECTORY files and not FILES. This problem occurs because in each frame the postconditions are made **strong**. NON-DIRECTORY-FILE from the postcondition set does not match *file*† (or the meaning representation that it is parsed into) from the user query. Thus the above query may not be processed correctly by the natural language understander. We need a rule to weaken the system reference to NON-DIRECTORY-FILE so that it becomes FILE. This is done by inferring non-directory-files to be files from the object hierarchy.

Another problem arises with the query ''How do I print a plain file?''. As preferred conditions are stored in frames, there will only be mention of FILEs in the precondition set for the PRINT frame. In any frame we make the preconditions as **weak** as possible. The frame selection process may mistakenly reject the PRINT frame. An inference rule is needed to strengthen the system reference to FILE so that it will match plain file. This problem is the complement of that above. In this case the user query has stronger information (*plain file*) whereas above it had

† Lowercase italicized characters are used to denote information from a user query.

weaker information (*file*). There is a requirement for an inference rule which will strengthen the system reference to FILE so that it becomes PLAIN-FILE.

Another type of problem occurs when more than one action or command is referenced in a user query. For example, in the query, ‘‘How do I find the misspellings in a file and then ‘more’ them?’’, the user has specified two concepts. The concepts *detecting-spelling-mistakes* and *moreing*∗ have been related together in this query. An inference rule is needed so that action frames from Transfer Semantics can be composed or interconnected in some way.

To summarize, there are three clear problems identified in the examples above: (1) sometimes postconditions for action frames are too strong, (2) sometimes preconditions for action frames are too weak, and (3) sometimes one frame is not enough to handle a query. Transfer Semantics will not work without inference rules. There is no requirement to define specific rules for every example of these problems. Any rules we develop will have to be general enough to cater for numerous natural language examples of the problems above. Let’s define some inference rules to take care of the above problems and some others.

4.2. A language for inference rules

It has already been decided that there need to be some rules which the system will use as a guide to selecting information from the object hierarchy. We need a language to define these rules. Of course, any language we define will only be an aid to describing the inference processes involved. The computer program would probably have all the rules implemented explicitly in functions or routines and would not need to worry about the language itself.

∗ ‘More’ is a command from UNIX which produces formated output on the screen.

There are many languages possible while defining inference rules. There are even more ways of implementing the rules once they have been defined. Axiomatic semantic techniques have been applied in exploring the logical foundations of computer programming. Axiomatic semantics seems a most lucid and explanatory method for defining our rules. We can construct abstract formalizations of inference in the spirit of axiomatic semantics. First, lets discuss the foundations of axiomatic semantics and get used to some notation.

Axiomatic semantics has been used in the formal specification of the syntax and semantics of computer programming languages. The paper by Hoare (1969) is a classic reference on the core ideas of axiomatic semantics. Many of Hoare's ideas were stimulated from a paper by Floyd (1967). A more mathematical description of axiomatic semantics, and particularly program verification, is described in Stanat & McAllister (1977). Other discussions are found in Hoare & Wirth (1973) and Algaić and Arbib (1978). Owicki and Gries (1976a, 1976b) apply the approach to parallel programming. A good introduction to the semantics is formulated by Pagan (1981).

An axiomatic semantics for programming languages will be sufficiently defined if the specifications enable one to prove any true statement about the effect of executing any program or program segment. There is also the requirement that the specifications do not allow the proof of any false statements. Specifications are analogous to the axioms and rules of inference from a logical calculus. Each specification describes a minimal set of constraints that any implementation of the subject language must satisfy. Computer programmers have used axiomatic semantics to construct proofs that programs possess various formal properties. Logical expressions are used to make assertions about the values of one or more program variables or the relationships between these values.

The class of assertions include formulae of the form,

{P} A {Q}

where P and Q are logical expressions, and A is a construct or statement from the subject language. The notation above is interpreted to mean that, ‘‘if P is true before the execution of A and if the execution of A terminates, then Q is true after the termination of A’’. P is called the *precondition* of the assertion and Q the *postcondition*. Any assertion of the form {P} A {Q} will be either true or false. It is assumed that a program will terminate after the execution of any A. Axiom schemata can be developed for various constructs in the language. Rules of inference (proof or deduction rules) enable the truth of certain assertions to be deduced from the truth of others. A rule of inference of the form shown in Figure 4.1 with $H_{1,} H_{2,} .... \; H_n$ being general assertions means that, ‘‘given $H_{1,} H_{2,} .... \; H_n$ are true, then H may be deduced to be true’’. This is called the *first principle of deduction*.

$$\frac{H_{1,} H_{2,} .... \; H_n}{H}$$

Figure 4.1. Definition of the first principle of deduction.

Also we define a rule of inference of the form shown in Figure 4.2 which means that, ‘‘if H_{n+1} can be deduced by assuming the truth of $H_{1,} H_{2,} .... \; H_n$, then H may be deduced to be true.’’ then H may be deduced to be true’’. This is called the *second principle of deduction*.

$$\frac{H_{1,} H_{2,} \ldots.\ H_n \mid\!- H_{n+1}}{H}$$

Figure 4.2. Definition of the second principle of deduction.

These rules of inference are independent of any particular domain under description. It is possible to build an axiomatic semantics for a programming language by defining many specific rules of inference. Some of the rules defined below have parallels with those for describing programming languages. First, let's define some useful notation.

4.3. A language for representing actions

We define a language for representing operating system actions or commands. The notation in Figure 4.3 is used to denote the fact that some user U can execute the action A to transfer the precondition set ({P}) to the postcondition set ({Q}).

$$\Big\{ \{P\}\, A\, \{Q\} \Big\} : U$$

Figure 4.3. A language for representing actions.

We call the information inside the bold braces (**{ }**) a *command environment*. The command environment describes the results of multiple or single commands. There may be many command environments existing in the system and many different users executing these. Also, any execution of a command environment will cause a state change in the system. Explicit objects within the precondition set {P}, or postcondition set {Q} shall be represented by lower case characters whereas actions, A, shall be represented by upper case characters. Trivially, if there are no preconditions imposed on some command the we write TRUE A {Q}. We also assume that the execution of action A does not have side effects which we do not know about. An example of a command environment for the COPY command is shown in Figure 4.4.

{ *{,,,file,,/usr/afzal/format,,} COPY {,,non-directory-file,,/usr/paul/papers,}* **}** :

User

Figure 4.4. A command environment for the COPY command.

We use commas to show that only some of the objects in condition sets are being made explicit. There may be many more. The named objects in precondition and postcondition sets refer to similar objects from the user query. These objects will also have definitions in the object hierarchy. If they do not then the system will not understand them. For clarity, we usually present the same referent as used by the user to denote objects. Of course, this is not what really happens as all queries are parsed into meaning structures. Also, the frames do not contain trivial objects for pre/postconditions, but constraints on objects. We do not show the relationships or constraints between objects in our notation. They are not needed to explain the salient ideas in this Chapter.

4.4. The first rule of consequence

One problem with Transfer Semantics is that conditions specified in the postcondition set are too strong to match user queries. There needs to be some method of weakening them. Let’s take a look at the problem query again. The user asked, “How do I print a file on the screen?” The problem was that any frame matcher couldn’t match *file* in the query (or whatever meaning representation it was parsed into) to NON-DIRECTORY-FILE file in the postcondition set for the PRINT action frame. We can use a rule of inference in unison with the object hierarchy to locate NON-DIRECTORY-FILEs as types of FILE. That is what we want, and the rule of inference is called the *First Rule of Consequence*. In general we have the formula shown in Figure 4.5.

$$\left\{ \frac{\{P\}\, A\, \{Q\}\,,\ Q \Rightarrow R}{\{P\}\, A\, \{R\}} \right\} : U$$

Figure 4.5. Definition of the first rule of consequence.

This general rule states that if {P} A {Q} is true and the postcondition Q implies R another postcondition, then the system can infer {P} A {R} to be true too. The system has derived a new frame <{P} A {R} U> by producing the postcondition set {R} from the postcondition set {Q}. More specifically, for the example noted above we get Figure 4.6.

$$\left\{ \frac{\{P\}\ \mathit{PRINT}\ \{,,\mathit{non\text{-}directory\text{-}file},,\}\ ,\ \mathit{non\text{-}directory\text{-}file} \Rightarrow \mathit{file}}{\{P\}\ \mathit{PRINT}\ \{,,\mathit{file},\}} \right\} : \mathit{User}$$

Figure 4.6. Application of the first rule of consequence.

The first rule of consequence is applied to the specific natural language form and we note that if the object frame FILE exists in the postcondition set, and NON-DIRECTORY-FILE implies FILE, then {P} PRINT {,,file,} is also true. Now, the new frame <{P} A {,,file,} U> will match the natural language query and the frame selector can choose the correct frame.

4.5. The second rule of consequence

Another problem with Transfer Semantics was that sometimes preconditions for frames are too weak. There needs to be some method of strengthening preconditions. Say the user asked, “How do I list a plain file?” The problem was that the precondition set for the LIST action frame only knows about FILEs and not PLAIN-FILEs. The frame selector may reject the listing frame. But, from an object frame hierarchy the system could have inferred a PLAIN-FILE to be a type of NON-DIRECTORY-FILE, and a NON-DIRECTORY-FILE to be a type of FILE. Then frame selection would work better. The rule of inference needed here is called the *Second Rule of Consequence*. The rule takes the general form shown in Figure 4.7.

$$\left\{ \frac{S => P\ ,\ \{P\}\ A\ \{Q\}}{\{S\}\ A\ \{Q\}} \right\} : U$$

Figure 4.7. Definition of the second rule of consequence.

This general rule describes that if another precondition S implies the precondition P, and {P} A {Q} is true, then the system can infer {S} A {Q} to be true too. The system has derived a new frame <{S} A {Q} U> by producing the precondition set {S} from the precondition set {P}. For the example problem we derive a specific formula shown in Figure 4.8.

$$\left\{ \frac{plain\text{-}file => file\ ,\ \{,,file,\}\ LIST\ \{Q\}}{\{,,plain\text{-}file,\}\ LIST\ \{Q\}} \right\} : User$$

Figure 4.8. Application of the second rule of consequence.

If the object frame FILE exists in the precondition set, and PLAIN-FILE implies FILE then {,,plain-file,} A {Q} is also true. It will be easier for the frame matcher to choose the PRINT frame now. Note that in this particular example we have applied the implication operator twice, i.e., plain-file => non-directory-file, and non-directory-file => file. Before we go on to discuss a very powerful inference rule there is a need to clarify some of the ideas above. There are two things we wish to clear up: (1) a question of inference direction, and (2) the meaning of the implication operator “=>”.

4.6. Inference direction and the meaning of implication

In applying the first rule of consequence we used non-directory-file => file, and in applying the second rule of consequence we used plain-file => file. However, there is a subtle difference in the way we did that for each rule. In the former case we already had NON-DIRECTORY-FILE as a postcondition in the frame and found FILE from that. Yet, in the latter case it was FILE that was in the frame. In the first case it’s easy to move up an object hierarchy from NON-DIRECTORY-FILE to FILE (stronger to weaker). In the second case how did we get, plain-file => file?, because non-directory-file => file, and directory-file => file, and device-file => file, and all those things that are types of file => file. This could have been done by finding all the objects that implied file until one matched with the user query. That’s all right for this example because we would only need to derive a handful of new frames. However, in any extended object hierarchy it may take forever to get the correct frame.

So, the way to do inference is to take the semantic representation of some object mentioned by the user (e.g., *plain file*) and to derive a relation between that and what exists in the frame. Of course we are lucky here because it turns out that what the user said was correct. Plain files can be a good precondition for printing. If the user specifies an incorrect precondition then no relation may exist and we will be stuck. Yet that is fine, because the user was wrong in the first place and we would tell him so.

Another point which needs clearing up is the meaning of the implication operator in the inference rules described above. What does it mean for non-directory-file => file? Intuitively, this means that a strong object always implies a weaker one if and only if those objects are related and of the same type. In the object hierarchy the relationship between non-directory-file and file is *type-of*. Therefore, if one object is a type of another, one implies the other. This would also be the case with an *instance-of* relation, but not with *part-of*. Implication is not commutative; device-file => file is true, this does not mean that file => device-file is also true. However, implication is transitive for type-of relations but not instance-of relations. If plain-file => non-directory-file and non-directory-file => file, then plain-file => file. A basis for implication within command environments has now been defined.

The implication operator ‘‘=>’’ is comparable with ISA (is a) and AKO (a-kind-of) links which have already been described in the field of knowledge representation. For example, Fass (1986a) describes such operations by demonstrating moves along ‘‘ancestor’’ paths in a semantic network. Other such descriptions are found in Bobrow & Winograd (1977), Brachman (1979) and Goldstein & Roberts (1977).

4.7. A theory and representation of query embedding

Many queries about operating systems involve more than one action to complete some process. For example, the query, “How do I stop a listing of my directory, which is printing on the Imagen?” involves three actions: *removing*, *listing* and *printing*. We call such queries *embedded queries*. The previous query is an an example of *explicit embedding* where three actions are explicitly mentioned.

It is possible to define a language for describing embedded commands or actions. We use the notation $[A_1 < A_2 < \ldots A_n]$ to denote an embedding set where action A_1 is embedded inside action A_2, and so on. One can think of embedding in terms of a stack where A_n is pushed on top of A_{n-1} and so on. Interpreting the stack, the postcondition {Q} from performing A_1 is passed as a precondition to A_2 and so on until we reach the top of the stack. For the previous query we have the embedding set, [LIST < PRINT < REMOVE] and for the query, “How do I print a listing of my directory on the Imagen?” we get, [LIST < PRINT]. In the latter example a directory is initially listed and then printed. In effect, the concept of listing is embedded inside printing. Certainly, in order to interpret queries involving embedding, we need to use some other inference rule to process action frames.

4.8. A rule of composition

As seen in the previous section a third power problem with Transfer Semantics is that sometimes people like to mention more than one action in a query. It is necessary to have an inference rule which concatenates or composes action frames together. If this is not the case then the frame selection mechanism may try to select between two different frames which are both relevant to the query. The rule for linking frames together is called the *Rule of Composition*. The general form for the rule of composition is given in Figure 4.9.

This general formula states that if {P} A_1 {Q} is true, and {Q} A_2 {R} is also true then we can infer {P} [$A_1 < A_2$] {R} to be true too. In effect, this rule specifies that the postcondition set found by applying a number of actions in sequence will be the postcondition set derived by applying the postconditions of any action in the sequence as preconditions to a subsequent action. It is important to note here that {Q} may only represent a subset of the total postconditions of A_1 or total preconditions for A_2. Also, the rules of consequence may need to be applied to show up similarities between the postcondition for A_1 and the precondition for A_2. A more

$$\left\{ \frac{\{P\}\ A_1\ \{Q\}\ ,\ \{Q\}\ A_2\ \{R\}}{\{P\}\ [\mathrm{A}_1 < \mathrm{A}_2]\ \{R\}} \right\} : U$$

Figure 4.9. Definition of the rule of composition.

specific formula for the example query, ‘‘How do I detect misspellings in a file and more them?’’ is given in Figure 4.10.

$$\left\{ \frac{\{P\}\ SPELL\ \{Q\}\ ,\ \{Q\}\ PRINT\ \{R\}}{\{P\}\ [\mathrm{SPELL} < \mathrm{PRINT}]\ \{R\}} \right\} : User$$

Figure 4.10. Application of the rule of composition.

From the above specific inference rule we deduce that if the postcondition of the action frame SPELL is applied as the precondition to PRINT, then it is inferred that the postcondition of PRINT is the postcondition of executing both actions. It is easy to think of the rule of composition as describing a mechanism which processes many objects and many actions. The rule is a Many Object-Many Action definition.

The rule of composition is an abstract representation of a mechanism for composing various commands. It is powerful because it allows us to compose command environments. We will term environments with more than one command *multi-*

command environments. It is possible to build a structure representing the state of the system at any time by concatenating command environments. We will use the term *system environment* to describe a complete user session (all commands and objects used) with an operating system. System environments are multi-command environments in the extreme. The rule of composition acts as a generator of multi-command environments. It can therefore be used by the natural language understander to understand any plans the user may mention implicitly in a query. We shall call the plan understander *PlanCon*. PlanCon will embody the rule of composition and the two rules of consequence.

Concatenated command environments can be produced dynamically by PlanCon on request. Some user may wish to know what happens to a number of objects after applying a number of actions. Using PlanCon OSCON could construct the state of the users view of some system involving any sequence of commands. The user or OSCON would be able to determine if the sequence in mind was productive or detrimental. PlanCon will enable OSCON to build representations of the state of some simulated environment envisaged by a user who is asking queries in a dialogue or context mechanism. A good description of such a mechanism is discussed by Arens (1986).

Now we have taken care of three very visible problems that Transfer Semantics would have without inference. Three more inference rules will be defined for completeness. These are called the AND, OR and No-consequence rules.

4.9. The AND rule

The AND rule specifies a conjunction of constraints which may be necessary for some action. Here's an example query where the AND rule would be applied: ‘‘How do I append the file mbox to /usr/paul/post?’’ In this case the user wants to append one file to another. Let's not worry for now about how the system knows that */usr/paul/post* is a file. The system needs to AND each file as a precondition to the APPEND action frame. The general form of the AND rule is shown in Figure 4.11.

$$\left\{ \frac{\{P\}\, A\, \{Q\}\,,\, \{P'\}\, A\, \{Q'\}}{\{P \wedge P'\}\, A\, \{Q \wedge Q'\}} \right\} : User$$

Figure 4.11. Definition of the AND rule.

This general formula states that if {P} A {Q} is true and {P'} A {Q'} is true then it is possible to infer {P ∧ P'} and {Q ∧ Q'} to be true too. We show a more specific formula for the example query above in Figure 4.12.

$$\left\{ \frac{\{,,,mbox,,,\}\ APPEND\ \{Q\}\,,\, \{,/usr/paul/post,,\}\ APPEND\ \{R\}}{\{,,,mbox,, \wedge\ ,,/usr/paul/post,\}\ APPEND\ \{Q \wedge R\}} \right\} : User$$

Figure 4.12. Application of the AND rule.

From the above specific inference rule we deduce that if the preconditions *mbox,* and */usr/paul/post* are to be applied to the APPEND action frame, then these preconditions can be ANDed together and applied at once. In the example above we have included, for clarity, the actual names of the copied files. Of course, in reality the file names are parsed into a meaning representation and the system would determine the types of these files. They may both be PLAIN, or one PLAIN the other NON-DIRECTORY and so on. Naturally, other processes are used to determine the type of a file. It is possible to think of the AND rule as processing many objects through one action. This is a Many Object-Single Action definition.

4.10. The OR rule

The OR rule specifies the disjunction of a number of preconditions for some action. These preconditions will produce a set of disjoined postconditions. An example query where the OR rule is applied would be, “How do I delete the files mbox and .mailrc?” In this case the user wants to know how to delete two files rather than one. The system can OR representations of the two files as preconditions to the DELETE action frame. The general form for the OR rule is given in Figure 4.13.

$$\left\{ \frac{\{P\}\,A\,\{Q\}\,,\,\{P'\}\,A\,\{Q'\}}{\{P \vee P'\}\,A\,\{Q \vee Q'\}} \right\} : U$$

Figure 4.13. Definition of the OR rule.

This general formula states that if {P} A {P'} is true and {Q} A {Q'} is true, then it is possible to infer {P $\vee$ P'} and {Q $\vee$ Q'} to be true too. A more specific formula for the example query is given in Figure 4.14. From the above specific

$$\left\{ \frac{\{,,,mbox,\}\ DELETE\ \{Q\}\ ,\ \{,.mailrc,,\}\ DELETE\ \{Q'\}}{\{,,,mbox,\ \vee\ ,.mailrc,,\}\ DELETE\ \{\ Q \vee Q'\}} \right\} : User$$

Figure 4.14. Application of the OR rule.

inference rule we deduce that, if the preconditions *mbox* and *.mailrc* are applicable to the DELETE action frame, then these preconditions can be ORed together and applied at once. This will save using the same frame twice. It is also possible to think of the OR rule as processing many objects through one action. This is another Many Object-Single Action definition.

4.11. The distinction of AND and OR

The AND and OR rules are distinct from the three rules described earlier. The AND and OR rules are used to add information together for a given frame. They do not derive new information to be placed in a frame. There is a difference between AND and OR rules in that the AND rule is defined because it is *necessary*, whereas the OR rule is defined for *efficiency* reasons. The function of the AND rule is to add necessary constraints on objects together for a frame. The OR rule is used to OR object constraints together in one frame which could have been processed separately by two runs of the same frame. This could be done with the rule of composition. By this we mean that the OR rule could be removed so that some parallel rule of composition executes an action or command over many objects concurrently. This could not be done with the AND rule as some frames such as APPEND **need** source and destination files to exist before execution.

In some operating systems it doesn’t matter if /usr/paul/post or mbox don’t exist before executing an APPEND command. However, in some systems it does matter, and as was said before, we are taking a general approach to operating system design.

4.12. The No-consequence rule

Trivially, the No-consequence rule is a ‘‘do-nothing’’ statement and is defined by Figure 4.15.

$$\left\{ \{P\}\ A\ \{P\} \right\}$$

Figure 4.15. Definition of the no-consequence rule.

The rule shows us that after executing an action A some preconditions do not change at all. A command like *date* in the UNIX operating system could be considered under a no-consequence rule because it does not really change the states of files and directories in the system. Any no-consequence rule can be executed a number of times throughout any command environment without having any effect on file or directory objects within that command environment. We must be careful while applying the no-consequence rule as certain objects such as the terminal screen will be changed with application of commands such as ‘‘date’’.

It is important to realize that the no-consequence rule is truly an element of Transfer Semantics. The no-consequence rule does transfer objects from one state to another where the new state is the same as the old one. Therefore, when PlanCon sees certain no-consequence commands it just applies the no-consequence rule and does not change objects like files and directories in the precondition set although the postcondition set may be changed. It turns out that there are commands in some operating systems which can be concatenated in a multi-command environment to simulate the no-consequence rule. For example, commands can have their effects reversed if they are followed by certain other commands.

In summary we have defined six rules of inference which can describe operations on action frames. The rule of composition is the only rule which involves multiple actions. All the other rules relate to single actions. Some may argue that the first and second rules of consequence are not really inference rules at all. Certainly some of the other rules **are** inference rules. It is our belief that objects such as *plain files* should not be stored as plain files but as types of file. Such information can be located in an object hierarchy and that process is called inferencing.

In building the understander we try to store as little information as possible and derive new information when it is needed. That is our *minimum-storage principle*. It may be the case that some combination of the six inference rules is needed in order to build domain-specific representations to match a user query. The order of application of combinations of inference rules may be important. This will need to be investigated.

4.13. A justification of the minimal-storage principle

We hope to include a learning component called *LeCon* in the understander at a later stage. This component will be similar to the component called UTeacher in the UC system (Wilensky et al., 1986). The component would allow any user to update knowledge in the understander through a natural language interface. Knowledge that could be updated includes adding new action frames or object frames or updating existing ones. Inference rules could be updated or new ones added to the inference database. We are not so naive as to believe that six is a magic inference number at all. In any case for an update system it would be nice if all knowledge was kept in the same place in the understander. Then, we must justify why we chose to have three types of knowledge in different places which may hamper updating. That is, why did we choose to separate out knowledge into object frames, action frames, and inference rules?

We believe that it is easier to keep the static data representation (action frames) small and use information from another static representation (object frames) together with inference rules to expand the scope of action frames. The system will become more efficient, as it is easier to match small frames containing localized information, and infer on that local information, than to search through large frames. We have come to one major conclusion while developing this research. In general we argue that [action-frames + object-hierarchy + inference-rules] is better than [ACTION-FRAMES∗] or even [ACTION-FRAMES + object-hierarchy] for any system.

4.14. Other work on representing inference

The action frames for Transfer Semantics are described as plans in much of the literature (see Fikes & Nilsson, 1971; Carberry, 1983). Carberry (1983) describes plans containing preconditions, partially ordered actions and effects. This is also a good description of our action frames. The rule of composition for building multi-command environments is similar to what Carberry calls *global plan context*. Each individual command environment which may be used to construct a multi-command

∗ By using upper case characters we hope to emphasize that terms refer to action frames containing a large number of conditions.

environment is what Carberry calls a *local plan context*. Carberry describes her work in the wider context of dialogue understanding and we hope to apply the rule of composition in this area. Kautz & Allen (1986) have defined a structure for modeling concurrent actions.

The UNIX Consultant (UC) program (Wilensky et al. 1984, 1986) has various elements of inference embedded within it. The UC system is divided into various components. The components called PAGAN (Plan And Goal ANalyzer) and UCPlanner involve procedures closely related to what we talk about in this Chapter. The PAGAN program hypothesizes the plans and goals under which some user is operating. PAGANs knowledge representation involves *planfors*. These are relations between goals, and plans for achieving those goals. Each plan is a sequence of steps. Therefore, plans in the PAGAN component can be compared to multi-command environments in OSCON where a number of command environments are concatenated to produce some effect. We differ with the UC approach on one issue as in PlanCon goals and plans are generated dynamically. Each plan or command environment is generated dynamically using rules of inference over action frames and input text. That is exactly why we need the inference rules described above. Yet, in PAGAN the steps of all plans are already stored statically in memory in a planfor database.

UCPlanner has the function of determining a fact that the user would like to know. The domain planner tries to determine how to accomplish a task, using knowledge about UNIX and knowledge about the user's likely goals. UCPlanner is a knowledge based common-sense planner. The planner creates plans for the user's UNIX goals. A goal detector is used to detect various goals that are necessary to complete in order to execute some user goal. Goals may be detected automatically. For example, new goals may be detected during the projection of possible plans.

This will happen if the planner notices some plan would fail when some condition is not satisfied. A new goal would be produced by the planner to satisfy the condition. Other goals that may be detected include background goals such as access to files. The goal detector finds goal conflicts such as deleting files which have protection. Stored plans exist in the system and these are similar to the action frames from Transfer Semantics.

In UCPlanner plans are selected and this involves two processes: (1) new plans can be derived, and (2) a process of plan specification fills in each general plan with more specific information. A process called projection is used to test whether a given plan will execute successfully. This is a test for possible problems in the plan: (1) conditions to be satisfied, and (2) possible goal conflicts to be resolved because of the effects of that plan. This involves three processes. The planner contains defaults to help in simulating some plan. These defaults may not be supplied by the user. Defaults would be to assume such things as files being text unless otherwise specified. Other processes include condition checking to ensure that plan conditions are satisfied in the system, and new goal detection where effects may arise which are not part of the user’s goals.

In PlanCon new plans are derived using the rules of inference and general plans can be filled with more specific information from the object hierarchy. The process called project in UCPlanner is similar to the command-environment generator mentioned above. We are in agreement with Wilensky et al. (1986, p. 50): ‘‘However, to answer more interesting problems it is necessary to be able to build new plans from existing plans. It would be impossible and undesirable to index an appropriate plan for each of the possible queries that a user might have.’’ That is exactly why we use inference rules in PlanCon.

The SINIX Consultant discussed by Kemke (1986) contains a rich knowledge base similar to that Transfer Semantics. Like Transfer Semantics the SINIX knowledge base consists of a taxonomical hierarchy of concepts. The leaves of the hierarchy correspond to SINIX objects or commands. Higher level concepts reflect more general actions or objects. In Kemke (1987) we are told that a Plan Generator will use a formal semantics of commands. Kemke says (p. 218), ‘‘The formal semantics description should be able to be used by a *Plan Generator* in order to construct ‘‘complex actions’’, i.e. plans, if the desired state or action specified in the user’s question cannot be realized using a single command but, instead, through a sequence of commands.’’ She talks of being able to describe the effects of commands by using a set of *primitive* or *basic* actions. That is exactly what we hope PlanCon does using the Rule of Composition.

The COUSIN system developed by Hayes (1982) and Hayes & Szekely (1983) has interesting similarities to our work. COUSIN can provide dynamically generated, contextually sensitive explanations about the current state of user interaction with the system. COUSIN only generates these dynamic help frames if the user makes a request for help without giving the name of some static knowledge frame. We can do this by using the rule of composition. That rule generates any command environment by concatenating or interconnecting individual command environments. As COUSIN is a command-level interface, each stage of user interaction will be executable, whereas with PlanCon command environments are representations used by the system to understand user queries. They are representations of what *would* happen if the user executed certain commands.

Sandewall & Ronnquist (1986) define a representation for action structures very similar to our own. Each action structure is defined in terms of *precondition, postcondition* and *prevail* conditions. Prevail conditions must hold for the duration of some action. An action structure (multi-command environment for us) is viewed as a set of actions (single-command environment for us). Each action has a start point and an end point. These would be the preconditions and postconditions in any multi-command environment. They have done an interesting job on how to formalize sequences of connected actions. This will be useful for doing parallel command operations. It is easy in UNIX to be doing one thing while numerous other processes are going on. Such processes are called background or child processes.

Alterman (1986) describes an adaptive planner which takes advantage of the details associated with specific plans. The planner still maintains the flexibility of a planner that works from general plans. Alterman tells us, ‘‘A planner that has access to general plans (alternatively abstract or high-level plans) is flexible because such plans will apply to a large number of situations’’ (p. 65). That is exactly why we have defined the rules of inference at a general level in the discussion above. Alterman makes a very important point that if we use a general planner then plans must be recomputed for specific plans and if one uses a specific planner there is a wealth of detail and there are problems with flexibility. His adaptive planner uses information from both specific and general plans.

Alterman’s approach is different from our own. He believes that specific plans should be stored and called up on demand. Then the specific plans can be tuned to the particular situation or context at hand. However, we believe that is easier to compute specific plans on demand rather than to store them. We do that because there are so many specific plans that it is impossible to store them all, or even a good enough set, to cover enough situations. Again, this is our *minimal-storage principle*. Alterman defines a process of abstraction which removes details form a plan. This is similar to our process of weakening postconditions for some plan. He also describes specialization as moving from a more abstract plan towards more specific examples.

This is similar to strengthening preconditions in PlanCon. Adaptive planning is in effect the application of the first and second rules of consequence and the AND and OR rules. The first rule of consequence defines specialization while the second rule defines abstraction. However in PlanCon we define preconditions to be weak and postconditions to be strong. We gave good arguments for doing that. This useful distinction is not made by Alterman.

An interesting discussion on hierarchical representations of causal knowledge is found in Gabrielan & Stickney (1987). They define a formalism for hierarchical causal models which provides explicit representations for time and probabilities. A system is defined in terms of a set of states and transitions among those states. A state is considered to be a complete or partial description of the system at a moment in time. Transitions define how changes in the system occur. An explicit representation of a transition or action can be defined explicitly in terms of start states (preconditions) and end states (postconditions). They introduce a number of formal definitions to construct a precise formulation of a hierarchical causal model. In their formulation more than one state can be active at any one time and many parallel transitions can occur simultaneously and asynchronously. This is all related to the command environment generator.

Minsky (1975) also notes that we need some method of applying transformations between frames in a system. He says, “I do not understand the limitations of what can be done by simple processes working on frames. One could surely invent some “inference-frame technique” that could be used to rearrange terminals of other frames so as to simulate deductive logic” (p. 229). We use the rule of composition to build new structures from action frames already existing in the system. Our inference rules are similar to what Minsky (1975) calls “formal operations”. He defines formal operations as, “...processes that can examine and criticize our earlier representations (be they frame-like or whatever).'' (p. 230)

Elements of the rule of consequence have also been described as ``enablement'' by Pollack (1986). Her example demonstrates that in a mail system some user may type HEADER 15 and this *enables* the *generation* of deleting the fifteenth message by typing ``DEL .''. This happens because typing HEADER 15 makes message fifteen the current message to which ``.'' refers. Pollack only considers what she calls *simple plans* which are a restricted subset of plans. Simple plans are those plans where the agent believes that all the actions in a plan play a role by generating another action. That is, the plan includes no actions that the user believes are related to each other by enablement. Simple plans can easily be generated by PlanCon.

4.15. Inference, parallelism and beliefs

In this chapter we have described inference rules which will solve some of the problems in Transfer Semantics. We showed that a knowledge representation for operating systems called Transfer Semantics will not work without inference. This was done by providing natural language forms that could not be processed by the natural language understander. The next step was to describe some general rules of inference that could be applied to action frames so that the frames would work for each of the these forms. Those inference rules were called (1) The First Rule of Consequence, (2) The Second Rule of Consequence, and (3) The Rule of Composition. We defined three more rules called the AND, OR and No-consequence rules. The AND and OR rules provide necessary and efficiency requirements for frames. The no-consequence rule allows us to specify commands which will have no real effect on objects in the command environments. The language of each inference rule has been borrowed from axiomatic semantics. This semantics has been used to provide formal descriptions of programming languages. We chose that semantics because of its clarity and coverage.

It is a significant feature of Transfer Semantics that there exists a number of inference rules enabling manipulation of action frames. Therefore, by using the object frame hierarchy and these inference rules an action frame can circumscribe a large quantity of domain-specific relations. We have shown the usefulness of inference rules of consequence and composition. The consequence rules enable the system to infer more detailed or less specific objects from an object hierarchy. Embedded queries involving many concepts can be interpreted effectively on application of the composition rule.

It is concluded that the six general rules of inference are necessary in expanding the scope of Transfer Semantics. Remember, the problem with Transfer Semantics is that only preferred conditions are specified in frames. The inference rules allow the system to infer stronger and weaker forms of preferred objects that are not represented in frames. However, we have not tackled the problem of the ability of people to ask questions about actions running in parallel. The system needs some understanding of parallelism. This will be discussed in Chapter 5.

In this Chapter we have not worried about problems of the distinction between what the user and system believes. That distinction is discussed by Pollack (1986) in a paper on plan inference. She proposes that models of plan inference in conversation must include this distinction. If this does not happen plan inference will fail as will the communication that it is meant to support. Pollack has implemented a plan inference model in SPIRIT, which is a small demonstration system that answers questions about computer mail. She makes a neat distinction between *act-types* and *actions*. Act-types are types of actions and correspond to what we call action frames. Actions correspond to specific actions to achieve some act-type. For example, *cat* and *more* are actions specified in our PRINT frame where the frame itself can be thought of as an act-type. Wilks & Ballim (1987) have proposed a first implementation of a ‘‘belief engine’’ called *ViewGen* that contains heuristics for the default ascription of belief. We hope to include an instantiation of this belief engine within the understander to model the interaction of system and user planning on the basis of differing beliefs and plans.

Chapter 5: Planning in parallel

In Chapter 4 we talked about a plan generator called PlanCon which is used in understanding plans appearing in user queries. An important inference rule was defined to take care of many action queries. That rule was called the rule of composition. In effect the rule acts as a plan generator which concatenates actions together in various ways. Queries like, ‘‘How do I nroff a file and then print it on the Imagen?’’ are understood using the plan generator. Although a general rule of composition was defined, there are many different ways it can be used. This is not surprising as this rule is very significant in any system. In Chapter 4 the rule of composition dealt with sequential action sets. However, when people ask questions about operating systems they mention the execution of parallel actions. Commands running in parallel in multi-tasking operating systems such as UNIX can be formalized with some new rules. Let’s recap on the Rule of Composition and continue to define some new rules.

5.1. A recap on the rule of composition

Sometimes people ask questions about operating systems where more than one action is mentioned. For example, a user may ask, ‘‘How do I detect misspellings in a file and more them?’’ Here, the user has mentioned two actions, one for *detecting-spellings* and the other for *printing* them. We defined a rule of composition which prevents the frame selection mechanism from selecting between two different frames where it should have selected both. The general form for the rule of composition is as shown below in Figure 5.1.

$$\left\{ \frac{\{P\}\ A_1\ \{Q\}\ ,\ \{Q\}\ A_2\ \{R\}}{\{P\}\ [\mathrm{A}_1 < \mathrm{A}_2]\ \{R\}} \right\} : U$$

Figure 5.1. A recap on the definition of composition.

This general formula states that if {P} A_1 {Q} is true, and {Q} A_2 {R} is also true, then we can infer {P} [A_1 < A_2] {R} to be true, too. In effect, this rule specifies that the postcondition set found by applying a number of actions in sequence will be the postcondition set derived by applying the postconditions of any action in the sequence as preconditions to a subsequent action. A more specific formula for the query, “How do I print a listing of system users on the laser printer?” is shown in Figure 5.2.

$$\left\{ \frac{\{P\}\ WHO\ \{Q\}\ ,\ \{Q\}\ PRINT\ \{R\}}{\{P\}\ [\mathrm{WHO} < \mathrm{PRINT}]\ \{R\}} \right\} : User$$

Figure 5.2. Applying the definition of composition.

This formula tells us that if the postcondition of the action frame WHO is applied as the precondition to PRINT then it is inferred that the postcondition of PRINT is the postcondition of executing both actions. In this case we had two actions to deal with. Sometimes there may be three or more actions mentioned in a user query.

When more than two actions are mentioned in a query we must recursively apply the rule of composition. Say for example, we have the query, ‘‘How do I delete a listing of my directory from the printer queue?’’ Queries like this do not appear often, though we should cater for them. There are three actions involved here, *deleting*, *listing* and *printing*. The object or data being manipulated is a directory. Initially the listing action is applied to the directory, then printing, then deleting-from-printer. Again the theory of embedding is used to describe this sequence of commands. The embedding set for this query is *[(directory) LIST < PRINT < REMOVE]*. List is embedded inside print and print is embedded inside delete. They operate on directory. Figure 5.3 represents the operation of listing and printing on a directory.

$$\left\{ \frac{\{,,directory,,\}\ LIST\ \{Q\}\ ,\ \{Q\}\ PRINT\ \{R\}}{\{,,directory,\}\ [\mathrm{LIST} < \mathrm{PRINT}]\ \{R\}} \right\} : User$$

Figure 5.3. Applying composition to LISTING and PRINTING.

The above rule of inference specifies the use of the actions LIST and PRINT over a directory file. The REMOVE command must still be applied to achieve the complete effect of the sequence [{P} LIST {Q} PRINT {R} REMOVE {S}]. We show this in Figure 5.4.

$$\left\{ \frac{\{P\}\ PRINT\ \{Q\}\ ,\ \{Q\}\ REMOVE\ \{R\}}{\{P\}\ [\mathrm{PRINT} < \mathrm{REMOVE}]\ \{R\}} \right\} : User$$

Figure 5.4. Applying composition to PRINT and REMOVE.

Figure 5.4. shows that the postcondition {Q} from executing PRINT is passed as a precondition to REMOVE which gives the postcondition R. We note here that PlanCon must derive the correct postcondition {R}, from the REMOVE action frame. The rule of composition may be applied any number of times to run a whole sequence of commands.

5.2. Piping as composition

The UNIX operating system allows users to *pipe* commands together where the output of one command becomes the input of another. A string of commands hooked together is called a *pipeline*. For example the pipeline command, *Who | wc -l*, will answer the user query, ‘‘How do I find out how many people are logged in?’’ *Who* is a command which produces a list of users logged in, and *wc* (word count) with the -l option counts the number of lines in this list, which has one line for each user. Piped commands or pipelines can be represented by the Rule of Composition (ROC). The query, ‘‘How do I find out how many people are logged in?’’ can be represented by Figure 5.5.

It turns out that WHO is a command which does not have any required preconditions but produces a postcondition {Q} which is a list of users logged in. There

$$\left\{ \frac{\{true\}\ WHO\ \{Q\}\ ,\ \{Q\}\ WORD\text{-}COUNT\ \{R\}}{\{true\}\ [\mathrm{WHO} < \mathrm{WORD\text{-}COUNT}]\ \{R\}} \right\} : User$$

Figure 5.5. Applying composition to WHO and WORD-COUNT.

are no required preconditions because there is always at least one person logged in — the user himself. This is a good precondition to word-count which produces the postcondition {R}.

To demonstrate the powerful utility of pipelining we will show how four commands could be concatenated together. For example, the pipeline command below will produce on the laser printer a long listing of all the *usr* files whose name contains *paul*:

```
ls -l /usr | grep paul | sort +3nr | lpr
```

The files will be sorted in reverse numerical order by the fourth field of the list and printed on the laser printer. All this can be represented by the ROC. In fact, we show the quadruple concatenation below.

The three command environments in Figure 5.6 show what happens when the four commands LIST, GREP, SORT and PRINT are used together in a pipeline. The ROC must be applied three times to derive final postcondition set {T} from initial postcondition set {/usr}. We have a representation of the execution for {/usr} [LIST < GREP < SORT < PRINT] {T}.

$$\left\{ \frac{\{,/usr,\}\ LIST\ \{Q\}\ ,\ \{Q\}\ GREP\ \{R\}}{\{,/usr,\}\ [\mathrm{LIST} < \mathrm{GREP}]\ \{R\}} \right\} : User$$

$$\left\{ \frac{\{Q\}\ GREP\ \{R\}\ ,\ \{R\}\ SORT\ \{S\}}{\{Q\}\ [\mathrm{GREP} < \mathrm{SORT}]\ \{S\}} \right\} : User$$

$$\left\{ \frac{\{R\}\ SORT\ \{S\}\ ,\ \{S\}\ PRINT\ \{T\}}{\{R\}\ [\mathrm{SORT} < \mathrm{PRINT}]\ \{T\}} \right\} : User$$

Figure 5.6. Computing quadruple composition.

Except for the first and last commands in a sequence all commands are called *filters*. So, in the above example GREP and SORT are filters. By no accident we note that PRINT typically occurs last in any command environment, and LIST typically occurs first. This happens because the frame for LIST contains a command called *ls* which does not accept standard input∗. Also, PRINT contains the command *lpr* which does not write to standard output but to the laser printer.

5.3. Redirection as composition

Redirection of standard input and standard output is common in the UNIX operating system. A user may ask, “How do I list my files and put them in a file called paul’s-file?” This can be done by using the command *ls -l* > *paul’s-file*, and the **>**∗ operator redirects output to paul’s-file rather than the screen. The ROC to do

∗ “Standard input” is a UNIX term referring to the source where input to commands comes from. The source is defined, by default, to be from the terminal although this default can be changed.

∗ The redirection operator, “**>**” is bolded so as not to be confused with the embedding operator, “<”.

this is shown in Figure 5.7.

$$\left\{ \frac{\{,files,,\}\ LIST\ \{Q\}\ ,\ \{Q\} > \{R\}}{\{,files,,\}\ [\mathrm{LIST} < >]\ \{R\}} \right\} : User$$

Figure 5.7. Applying composition to redirection of input.

The symbol > although not really a command acts as one, and transfers standard output to another location. It is also possible to redirect standard input. For example, in the query, ‘‘How do I mail a file to Afzal?’’ the command, *mail afzal@nmsu* < *file* will do the job. Standard input is redirected to come from file rather than the terminal. The ROC also deals with redirection of standard input and this is shown in Figure 5.8.

$$\left\{ \frac{\{file\} < \{Q\}\ ,\ \{Q\}\ \mathrm{MAIL}\ \{R\}}{\{file\}\ [< <\ \mathrm{MAIL}]\ \{R\}} \right\} : User$$

Figure 5.8. Applying composition to redirection of output.

In Figure 5.8 we show how redirected standard input is passed to mail which produces a postcondition {R}. It seems that the ROC is capable of doing many types of sequencing and is very wide ranging. However, there are some operations which the ROC does not cater for yet.

5.4. Limitations of the rule of composition

The rule of composition is extensive yet will not handle the query, ‘‘How do I find a job number and then kill the job?’’ In this case the user needs to find a job number (one action or command) and then kill it (another action or command). However, the output of one command is not passed as the input to the other. The difference with this type of query is that the user has specified a concatenation of commands that are not usually concatenated together. The KILL frame just needs some of the information from the FINDING-JOB-NUMBERS frame.

We need to define a new inference rule, involving some type of composition, i.e., a variation on the rule of composition. The rule will specify sifting of information from one action to be passed to another. There will surely be many ways of sifting information from commands. One can easily see now that it would certainly be naive to suspect that six general rules of inference was enough. Our next job is to locate and specify new and interesting rules of inference.

5.5. A selective rule of composition

Say some user asks the question, ‘‘How do I find a process and kill it?’’ Here we have two actions mentioned in the input query, *finding-a-process* and *killing-a-process*. As the ROC stands it would compute something like Figure 5.9.

Of course this is incorrect as it isn’t the complete output of FIND-PROCESS that is passed to kill, but a selective piece which is the job number. Otherwise the rule would specify killing **all** processes. We must define the selection or filtering of output from certain commands which is used as input to other commands or actions. We define a new rule, shown in Figure 5.10, called the *selective rule of composition (ROC_1).*

$$\left\{ \frac{\{P\}\ \mathit{FIND\text{-}PROCESS}\ \{Q\}\ ,\ \{Q\}\ \mathit{KILL}\ \{R\}}{\{P\}\ [\text{FIND-PROCESS} < \text{KILL}]\ \{R\}} \right\} : \mathit{User}$$

Figure 5.9. Applying composition to FIND-PROCESS and KILL.

$$\left\{ \frac{\{P\}\ A_1\ \{Q\}\ ,\ \{Q\}_{\subset}\ A_2\ \{R\}}{\{P\}\ [\ \mathrm{A}_1 <_{\subset} \mathrm{A}_2\ ]\ \{R\}} \right\} : U$$

Figure 5.10. Definition of the selective rule of composition.

The general formula states that if {P} A_1 {Q} is true, and $\{Q\}_{\subset}$ A_2 {R} is also true then we can infer {P} [$A_1 <_{\subset} A_2$] {R} to be true too. The rule shows that it is only a subset of {Q} from A_1 that is passed to the second action A_2.

5.6. A composition rule for parallelism

Another problem occurs when a user wishes to execute background processes in some multi-tasking operating system. Each rule of composition defined so far has been purely sequential. The output of each command, or a subset of that is passed to the next command in the sequence.

Say some user asks the query, ‘‘How do I print a file in the background and edit a file in the foreground?’’ In that case the user is talking about running two actions in parallel. The rule of composition does not cater for this. We define a new rule of composition called the *parallel rule of composition* (ROC_2). This is shown in Figure 5.11.

$$\left\{ \frac{\{P\}\, A_{1_{||}}\ \{Q\}, \{P'\}\, A_{2_{||}}\ \{Q'\}}{\{P \vee P'\}\ [\ \mathrm{A}_1 <_{||} \mathrm{A}_2\ ]\ \{\ Q \vee Q'\}} \right\} : U$$

Figure 5.11. Definition of the parallel rule of composition.

In Figure 5.11 we show that the two actions A_1 and A_2 run in parallel and the preconditions for each action are ORed together because they both exist. This is also true for the postconditions {Q}. This rule of composition is very like the OR rule described in Chapter 4. Of course, the difference is that the OR rule is a single action rule, whereas ROC_2 is a double action rule, i.e., ROC_2 includes A_1 and A_2, whereas the OR rule only has A. Of course, if we have A_1 equal to A_2 in Figure 5.11 then ROC_2 translates to Figure 4.13 (OR rule).

Any command environment representing ROC_2 will have a sub environment running in parallel to the master environment. It may also be necessary to specify communication between data elements or objects in parallel executions although this is doubtful. For example, in UNIX it is not advisable to alter files or other objects in both background and foreground processes. We now have three rules of composition: ROC_0, ROC_1, and ROC_2. However, that is not enough.

5.7. A composition rule for forking

There is another concept for which we must define a rule of composition. Some user may wish to pass the output of a command to more than one location at one time. This is called *forking* and involves both parallelism and sequential operations. For example, a user may want to know, “How do I list a file both on the screen and into a directory file?” This can be done by, **ls | tee dirfile**. The command *tee* places the output of list in two locations, *dirfile* and the *standard output*. We need a new rule to handle forking and it is called the *fork rule of composition* (ROC_3). This rule is shown in Figure 5.12.

$$\left\{ \frac{\{P\}\,A\,\{Q\}\,,\{P\}\,A_{\Upsilon}\,\{R\}}{\{P\}\;[\;\mathrm{A} < \mathrm{A}_{\Upsilon}\;]\;\;\{Q \vee R\}} \right\} : U$$

Figure 5.12. Definition of the fork rule of composition.

This rule shows that executing any fork command in a sequence will cause forking of the preconditions {P} to give the postcondition in two locations {Q} and {R}.

5.8. A composition hierarchy

We have now defined four rules of composition. The first called ROC_0 was a general rule specifying no detail as to how commands were linked together. Next, we defined the selective rule of composition called ROC_1 which passes selective pieces of postconditions from one command too another. This was followed by the parallel rule of composition called ROC_2 which allowed the composition language to describe actions running in parallel. We further extended the depth of composition by defining the fork rule of composition (ROC_3). We show a hierarchy of composition in Figure 5.13.

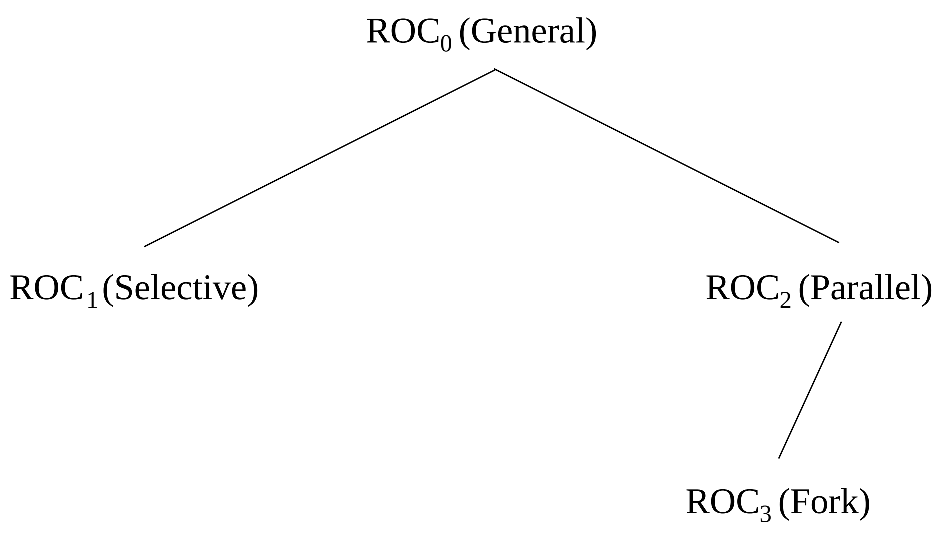

Figure 5.13. A graphic hierarchy of composition.

$ROC_{1,2,3}$ are an extension of the depth of composition rather than its breadth. We have defined each rule as there needs to be a definition of the specific **types** of composition in the system somewhere, and that is brought out by defining clearly more specific instances of composition. It is possible to prove that $ROC_{1,2,3}$ are all collapsible to ROC_0 under certain conditions. That is good because it shows that ROC_0 has in its definition the inherent elements of composition. It is possible to demonstrate subsumption by ROC_0 of the different rules.

First of all we show that ROC_2 subsumes ROC_3. Remember, ROC_2 described parallel composition and ROC_3 described forked composition. Taking Figure 5.11 denoting (ROC_2) we substitute {P} for $\{P'\}$ everywhere in the equation. We get Figure 5.14. Now, $\{P \vee P\}$ collapses to {P} and we get the fork rule of composition ROC_3.

$$\left\{ \frac{\{P\}\, A_{1_{||}}\, \{Q\}, \{P\}\, A_{2_{||}}\, \{Q'\}}{\{P \vee P\}\, [\, \mathrm{A}_1 <_{||} \mathrm{A}_2 \,]\, \{\, Q \vee Q'\}} \right\} : U$$

Figure 5.14. Equivalence of parallel and fork composition.

It is also possible to show the parallel rule of composition (ROC_2) as a special case of the general rule (ROC_0). If we take Figure 5.11 and let $\{P'\}$ be {Q} everywhere in the equation then we find Figure 5.15. In effect, this means that the precondition of one of the parallel actions ($A_{1_{||}}$) is the same as the postcondition for the other action ($A_{1_{||}}$). In Figure 5.15 $\{P \vee Q\}$ can decompose to {P} and $\{Q \vee Q'\}$ can decompose to $\{Q'\}$. This would happen when two actions run in parallel and only one of the preconditions exists ({P}) and only one of the postconditions exists (Q'). We get the general rule of composition as shown in Figure 5.1.

Now, we have shown that ROC_3 is a special case of ROC_2 and ROC_2 is a special case of ROC_0. It follows that ROC_3 can also be a special case of ROC_0. Therefore we have shown that for certain cases $ROC_{2,3}$ are equivalent to ROC_0. Trivially, ROC_1 translates to ROC_0 when $\subset$ is $\subseteq$. Then $Q_{\subseteq}$ could be $Q_{=}$ i.e., $\{Q\}_{\subseteq}$ would be

$$\left\{ \frac{\{P\}\, A_{1_{||}}\, \{Q\}, \{Q\}\, A_{2_{||}}\, \{Q'\}}{\{P \vee Q\}\, [\, \mathrm{A}_1 <_{||} \mathrm{A}_2\,]\, \{\, Q \vee Q'\}} \right\} : U$$

Figure 5.15. Equivalence of parallel and general composition.

{Q}. The formula for ROC_1 then collapses to Figure 5.16. Figure 5.16 is equivalent to Figure 5.1 (ROC_0).

$$\left\{ \frac{\{P\}\, A_1\, \{Q\}\, , \{Q\}\, A_2\, \{R\}}{\{P\}\, [\, \mathrm{A}_1 < \mathrm{A}_2\,]\, \{R\}} \right\} : U$$

Figure 5.16. Equivalence of selective and general composition.

We must emphasize that the various rules of composition are a declarative representation of procedures in the system which manipulate action frames. In an inference module it would be necessary to also define a hierarchy including each ROC. The next question that must be answered is, how does the system know which rule to apply? The general rule of composition would be tried first, then probably selection, and then the parallel rules. That is just a conjecture and empirical studies may prove us wrong.

5.9. Other work on plan hierarchies and parallel planning

We have now defined a hierarchy of inference rules for the rule of composition and there are also have five other rules of inference. The hierarchy allows PlanCon to understand complex plans and goals in user queries. Hierarchies of component goals and actions for domain-dependent plans have been used by Carberry (1983). Litman and Allen (1984) and Pelavin and Allen (1987) have also developed a model based on a hierarchy of plans and metaplans.

Litman and Allen (1984) discuss the modeling of plans of speakers in task domains. They develop a model based on a hierarchy of plans and metaplans that accounts for clarification subdialogues and topic change. They consider plans as a network of actions and states connected by links indicating causality and subpart relations. Each plan has a *header,* which names the plan and *parameters* that exist in the header. Plans also have a set of *constraints,* which are assertions about the plan and its terms and parameters. Plans may also contain *prerequisites* (preconditions), *effects* (postconditions) and *decomposition* (sequences of actions). They give an interesting account of how plans may be linked together in a plan hierarchy which is useful for understanding sentences input to a contextual understander. They use a focus mechanism to direct the planner through a hierarchy of plans. It is our intent to add a focus mechanism to the system at a later date. The Litman and Allen planner is intended to be a more general planner for general language understanding whereas PlanCon is intended to be characterizing the specific operations that people can do with actions, and in particular how people can compose various actions.

Pelalvin and Allen (1987) describe a model for concurrent actions with temporal extent. They describe a semantic structure which provides a basis for defining a semantic structure for describing interactions between actions, both concurrent and sequential, and for composing simple actions to form complex ones. It also treats actions that are influenced by properties that exist and events that occur during the time that the action is to be executed. One theme of their approach is to capture what is happening while an event is occurring. They directly treat events affected by conditions that hold during execution. For example, the event of ‘‘sailing across the lake’’ is described which can only occur if the wind is blowing *while* the sailing is occurring. Composition is defined in terms of a formal logic notation.

Sandewall and Ronquist (1986) consider structures for actions which are partially ordered for time and which may occur in parallel. They show how concurrent actions can be dealt with by using a petri-net approach. They relate their work to theories and languages of concurrent programming. Again, this work relates to the function of PlanCon.

Chapter 6: Meaning representations

So far we have been discussing a knowledge representation for operating systems, and how this representation may be used to understand natural language queries. However, we have not discussed the meaning representation that English queries are parsed into, or how this meaning representation chooses the correct frames. The job of the parsing process is to parse natural language input into a good meaning representation. One of the first questions we must ask ourselves is what type of queries do people ask.

6.1. The nature of queries about operating systems

A good way to understand the requirements of a natural language understander for operating systems is to build a good theory of the way people use English to ask questions about these systems. When people ask questions about the UNIX operating system they often refer to a number of interrelated actions. Numerous objects are associated with these actions. For example people ask questions like, ‘‘How do I print a file on the Xerox with pageheaders?’’, or ‘‘How do I spell a file and then have the mistakes printed on the Imagen?’’. The former query has one action, that of *printing* and the latter has two actions that of *finding mistakes* and that of *printing-on-Imagens*. Of course, some queries have no actions at all. These are queries like, ‘‘What is a file?’’ or ‘‘What is a pipe?’’ Such queries are static and involve answers which are descriptive rather than dynamic.

Therefore, operating system queries are about the dynamics of the system, i.e., queries about actions such as printing, removing or deleting; or about the statics of the system, i.e., queries about static objects such as files, file-structure, pipes, and so on. In Section 4.7 we called the former *dynamic queries* and the latter *concept description queries*.

If dynamic queries did not exist, then it would be easier to build an operating system consultant. All we would have to do is to store a manual about operating systems in OSCON, and if a person asked a question about say, files the system would just print out any information about files. However, people ask questions like, ‘‘How do I print a file on the Imagen with pageheaders?’’ which involve actions such as *print*, and constraints on those actions such as *file* and *Imagen* and *pageheaders*. Therefore, it is necessary to develop some mechanism whereby meaning representations for multiple action queries can be integrated in a sensible manner. That is what this chapter is about.

6.2. A tutorial on the theory of embedding

We can assume that any query about operating systems includes a number of actions (which may be zero) and objects manipulated by those actions. Any meaning representation of a query must contain in some semantic form the actions and objects and how these are related together. Concept description queries have no actions at all. Dynamic queries contain one or more actions. Dynamic queries with one action are represented without much difficulty. In the meaning representation we can just include that action and any objects related to it.

Dynamic queries containing more than one action are more difficult to deal with. There needs to be some way of relating actions together. Let’s look at some more examples. In ‘‘How do I send a troff file to the Imagen?’’ there are two actions, *troffing-a-file* and then *sending-it-to-the-Imagen*. In the query, ‘‘How do I remove a file printing on the Imagen?’’ there are also two actions. Those are *printing-a-file* and then *removing-it*. What do we notice about the actions in each query? Each action is related temporally to other actions in the query. One action is executed before another and the ordering is important. Certain actions operate on objects and change their states. Other actions come along later in time and transfer object states into new ones. We think of natural language queries in terms of actions processing objects in time. A good way to represent such actions will be to keep that notion of time as it is important. Actions can be sequenced or embedded within one another. Actions are black boxes which take objects as input and produce new objects as output. Each set of actions is looked on as being an embedded set.

The theory of embedding seems to be a good one. To understand queries about operating systems we need to be able to recognize actions and objects in the input. We need to represent the meaning of words if the system is to determine whether words in input sentences are actions or objects. A mechanism should match words in the input to their meaning representations. That mechanism should determine whether words are actions or objects. Then there must be some processor which unifies or concatenates word representations together to build complete meaning representations of whole queries. Such complete meaning representations will capture the temporal relations between various actions and their objects.

6.3. The components of a meaning representation

To determine whether words in the input are objects or actions we need to represent the meaning of words in the system. Therefore we need a *dictionary* of words which tells us the type of each word. So, *print* will be represented as an action and so will *delete* and *move*. *Files* and *directories* will be typed as objects. We build a dictionary with entries like *(file location object)* and *(print action)*, *(directory location object)* and *(user actor)*. Notice that *file* and *directory* are also marked as being specific types of objects i.e., locations.

The next problem we must worry about is how to define allowable operations of actions. We should build some structures which represent various actions and how they relate to objects. Such structures would recognize incorrect operations of actions. A database of patterns is defined to represent legal action operations. Patterns are semantic templates for expected legal actions. We construct a pattern database with entries like *(observe-obj <person> <object>)*. This entry tells us that a legal sentence could include actors observing objects. Thus the query, ‘‘How do I see a file?’’ would match this pattern. We can call the patterns *action templates*.

To build meaning representations for sentences, we need to be able to link action templates together by some means. To do that there needs to be some precise definition of what each action template means. Each action template should have an associated meaning structure. Such meaning structures should include actions, objects related to actions, types of those objects, and actors. We call such representations *deep case structures* and one is shown in Figure 6.1.

```
(observe-obj  ((actor _*)
                (description _))
              ((object _)
                (description _))
              ((location _)
                (description _))
              (frames print list))
```

Figure 6.1. Case structure for observing objects.

∗ We use the symbol ‘‘_’’ to denote an unfilled slot in a case frame.

Figure 6.1 is a case structure which tells us that to observe an object there can be an observer, an object of observation, and a location for the observation to occur. Each entity may have some *description* tagged to it. The case structure also references two frames. These are the action frames in the system (discussed in Section 3.4.) that represent domain specific knowledge about operating systems. For this example the frames are print and list which specify printing-objects and listing-objects respectively. Already we note that domain specific information will be selected if this case structure is referenced.

6.4. Embedded action representations

As already mentioned, user queries about operating system commands contain embeddings of actions. It should be possible to create representations of nested or embedded deep case structures to describe interrelated actions. We call such representations embedded action representations (EREPs).

Operating system commands are related to each other in specific ways. When users ask questions about such actions they usually get these relations correct. Therefore, if we build EREPs from user queries, the EREPs should be a good approximation of the relations. This means that domain-specific structures produced from EREPs should often be correct. In fact, if the domain-specific structures are not correct the input query also contains relations which are incorrect. It is possible to build EREPs from queries about operating systems and to translate them into domain-specific representations.

Examples of typical actions which can occur in EREPs are printing, listing and deleting. People can ask questions about UNIX such as, ‘‘How do I print a listing of my directory?’’, or ‘‘I need to print a file.’’ In the former example we build an EREP where the concept *listing* is embedded inside the concept *print* and in the latter case *print* is embedded inside *need*. These are examples of double embedding. Yet, triple embeddings result from queries such as, ‘‘How do I delete a listing of my directory, printing on the Imagen?’’ We will now go on to show how EREPs can be used to understand natural language queries.

6.5. Null embedded queries ($\{A_i\}_{i=0,1}$)

If a query involves no actions, or just one action, then there will be no embedding at all. If we use A_i to represent the number of actions in a query then null embedded queries are denoted by $\{A_i\}_{i=0,1}$. Concept description queries are simple questions about objects with no presence of operating system actions. Therefore, concept description queries will always exhibit null embedding. It is also interesting to note that concept description queries do not include actors. The reason for that, of course, is that there is nothing for them to act upon. A typical example of a concept description query is the sentence, ‘‘What is read protection?’’. The action template *(be <object>)* is used in deciphering this query. *Protection* is defined under the category object from its dictionary entry i.e., *(protection object)*. The case structure for the *be* action template is instantiated to give the structure in Figure 6.2.

```
(be ((object PROTECTION*)
     (description READ))
    (frames protection))
```

Figure 6.2. Instantiated case structure for ‘be’.

Dynamic English queries illustrate null embedding when only one action is mentioned. For example, the query ‘‘How do I delete a file?’’ has a representation with no embedded actions at all. The query is parsed into the structure shown in Figure 6.3.

```
(delete-obj  (actor USER)
             ((object FILE)
              (description quantity ONE))
             (frames remove))
```

Figure 6.3. Instantiated case structure for ‘delete-obj’.

6.6. Positively embedded queries ($\{A_i\}_{i \geq 2}$)

There are many types of embedding present in meaning representations resulting from dynamic queries. We have already seen that dynamic queries exhibit null embedding. However they also exhibit positive embedding which means that the query includes more than one action. We call queries with two or more actions positively embedded queries and they are denoted by $\{A_i\}_{i \geq 2}$. Such queries have at least one positive embedding of one action inside another. Also, we have discovered that there are many types of positive embedding and there are many different ways of recognizing and processing these.

* In the following case structure diagrams capitalized items indicate values filled in from dictionary entries.

6.6.1. Explicit embedding

Explicit embedding occurs in representations for queries involving two or more actions. For example, the meaning representation for the query ‘‘How do I print a listing of my directory?’’, has the concept of listing embedded inside the concept of printing. In processing this query, an *observe-pat* case structure is instantiated to give Figure 6.4. An *observe-pat* case structure is selected because the system recognizes that the directory was first listed and then printed. The user wishes to observe an object which was already observed.

```
(observe-pat (actor USER)
             (case (observe-obj (actor _)
                                ((object DIRECTORY)
                                 (description quantity ONE))
                                (frames print list)))
                   (frames print list))
```

Figure 6.4. Meaning representation exhibiting explicit embedding.

Figure 6.4 shows that deep case representation for listing is nested inside the representation for printing. The inner case structure is filled out first, and contains directory as an object, because the actor is asking about listing directories. The actor is not filled in yet as the action template for observing an object does not find an actor in the phrase, *...listing of my directory*. The actor slot in the outer case representation is instantiated to be USER. This was found from, *How do I print..* where the actor was mentioned. In Figure 6.4 we note that the actor slot in the inner case structure is not instantiated. However, this information would be determined from the outer case structure and promoted inwards.

6.6.2. Implicit embedding

Some word in a user query may indicate implicitly the existence of another action although this action is not mentioned directly. Say, for example, the system is given the query, ‘‘How do I delete mail files?’’ Naively, the system would believe that the user just wants to delete an object called *file* with description *mail*. OSCON would overlook the fact that another action (in this case *mail*) has created the object. In deriving a meaning representation for this example a first step would be to construct the structure in Figure 6.5.

```
(delete-obj (actor USER)
            ((object FILE)
             (description quantity MORE-THAN-ONE)
             (description type MAIL))
            (frames remove))
```

Figure 6.5. Implicit embedding I.

Now, to solve the problem of not recognizing implicit embedding, each object or action could be checked every time a representation is produced to establish whether that object or action refers to another action template. In this case, *mail* is recognized as being another action. Indeed, *mail* (a description on the object *file*) references the action template *send* and its respective case structure. After some processing the EREP in Figure 6.6 is computed. It is noted that in Figure 6.6 the concept *send* is embedded inside the concept *remove*. The actor as user is expressed

```
(remove (actor USER)
        (case (send (actor USER)
                    ((object FILE)
                     (description quantity MORE-THAN-ONE))
                    (frames MAIL)))
            (frames remove))
```

Figure 6.6. Implicit embedding II.

inside each case structure. The send case structure represents the fact that the user wishes to remove more than one file from the quantity descriptor.

6.6.3. Shadowed embedding

Often actions such as *wanting* or *needing* can shadow the UNIX action which is more important for the system to locate. We call this shadowed embedding because a shadowing verb will enclose or shadow a verb about some UNIX action. Although we are primarily concerned with locating UNIX concepts, we do realize the importance of shadowing actions. Such actions are very useful in detecting the goals of the user (see Wilensky et al. 1984, p. 589). The direction of reading the input query is important because shadowing may occur while reading a sentence in one direction although it does not in the other. Examples of shadowing exist in sentences like “I would like to delete a file”, and “I need to print a file”. On reading the latter query from left to right *need* shadows *print*. However, if the query is read right to left we get, *A file, to print, I need?* In this case *print* is not shadowed. Yet, OSCON reads sentences left to right and therefore it needs to handle shadowing.

There are action templates for shadowing verbs in the pattern database such as, (s-verb <person> <pattern>). For the query, ‘‘I need to print a file’’ the meaning representation shown in Figure 6.7 is formed.

```
(s-verb (need) (actor USER)
               (case (observe-obj (actor USER)
                     ((object FILE)
                      (description ONE))
                     (frames print list))))
```

Figure 6.7. Shadowed embedding.

We note that the s-verb case structure has only one case slot other than *actor*, called *case*. The actual shadowing verb used in the sentence is tagged onto the EREP as it may be useful in later processing. For example, such information would be useful for discovering the intention of the user.

6.6.4. The intricacy of redundant embedding

Representations with redundant embedding are more a characteristic of the parsing strategy than a characteristic of English. For example, while parsing the query, ‘‘How do I use print to print a file?’’, the case structure for observing objects would become embedded within itself. This happens because of implicit embedding rules. In effect, (1) the user has mentioned printing files, and (2) the user has also mentioned the operation for doing so i.e., *print*. It would certainly be a mistake to embed in examples such as this and OSCON must have strategies to recognize redundant embedding. For this example the system produces the case structure in

Figure 6.8.

```
(s-verb (use) (actor USER)
              (object PRINT)
              (case (observe-pat observe-obj) (actor _) ...)
              (frames FRAME (object)))
```

Figure 6.8. Redundant embedding I.

From the previous example of implicit embedding we notice that the system would find PRINT and believe there should be another embedding of the *observe-obj* case structure. Yet, this is wrong because the case structure for observing objects already exists. There must be another rule which recognizes that implicit embedding is not carried out if there seems to be redundancy. Therefore a counter rule will dictate that PRINT does not call up another case structure. We must be careful in applying the counter rule too. For example, “How do I print listed files?” involves an embedding of *observe-obj* inside *observe-obj*. The inner case structure for listing is referenced again by implicit embedding techniques and the problem here is that we really do wish to embed. There seems no way out of all this. But, look again at the example of redundant embedding. We notice that the query contains the shadowing verb *use* and that is what the system needs to look for while applying the counter rule. The system will correctly represent the query, “How do I use print to print a file?” as Figure 6.9. It is noted that in Figure 6.9 that objects have been promoted inwards from the query. The clause “...to print a file” instantiates objects in the inner case structure. No frames are called forward by FRAME (object) because of the

```
(s-verb (use) (actor USER)
              (object PRINT)
         (case (observe-obj (actor USER)
                            ((object FILE)
                             (description quantity ONE))
                            (frames print list))))
              (frames NIL))
```

Figure 6.9. Redundant embedding II.

counteractive rule for redundancy. Note however, that in a query like ‘‘How do I use print?’’ FRAME (object) would call forward these frames as they are not referenced in any inner embedded case structure.

6.6.5. Negated embedding

Negation of concepts can arise in many queries. The query, ‘‘I can not delete my file’’, is an example. Usually, negation will occur with triple embedding in meaning representations. The meaning representation for the latter query is shown in Figure 6.10. In Figure 6.10 the case representation *remove* is embedded inside *not* and not is embedded inside a can shadowing case structure. There are no frames for the not case structure just like there are none for the s-verb can.

6.7. The selection of knowledge

As we can already see the case structures, and hence the EREPs, maintain references to various action frames. It is the job of a frame selector to work out the frame(s) most likely for the query in question. This is done by matching information

```
(s-verb (can) (actor USER)
        (case (not (actor USER)
                        (case (remove (actor USER)
                                      ((object FILE)
                                       (description owner USER)
                                       (description quantity ONE))
                                      (frames remove))))))
```

Figure 6.10. Negated embedding.

from the meaning representations to the frames and finding the frame with the maximum number of matches.

Preferences are used in frame selection processes where the frame with the maximum number of preferences satisfied is probably the best frame for interpreting the input. For example, the print frame will have more preferences satisfied than the list frame from the query, ‘‘How do I list a file on the Imagen?’’ Of course, that is because one usually associates Imagen printers with printing rather than listing.

It is important to note that only the best conditions are selected while matching a frame to an initial meaning representation of some query. For each condition we determine the ratio of matched to non-matched predicates. The best condition is the one with the highest ratio. For any condition to be best not all its preferences have to be satisfied. Indeed, we saw in Chapter 5 that the process of weakening postconditions and strengthening preconditions is required because local preferences in conditions are not satisfied.

6.8. Meaning representation and surface structure

We have discussed meaning representations but not how the system gets there from surface structure. The system will have a natural language parser as a front end to analyze natural language input. We intend to try out a number of parsers for the system and evaluate the performance of each. Shallow representations of English queries will be produced by the particular parser in use. Examples of such parsers are discussed in depth by Ball and Huang in Wilks (1986). We believe the process of understanding language to be semantic and knowledge-based as opposed to syntax-based and the algorithms that implement this view exploit a notion of computing the coherence of textual meaning.

One of our parsers exists as part of XTRA, (see Huang, 1985) a machine translation program, which uses a Semantic Definite Clause Grammar (see Pereira & Warren, 1985) and the semantics is a modification of that discussed in Wilks (1975b) coupled with new relaxation mechanisms. XTRA's distinctive features are its treatment of conjunctions, its phrase and clause attachment procedures (see Wilks, Huang & Fass, 1985) and its relaxation features from semantic constraints. The XTRA system is composed of two phases; parsing and generation. We are only concerned with the parser from XTRA. The system produces a syntactico-semantic tree for some input sentence. The format of the tree is borrowed from Bougarev (1979) though the approaches for getting the representation are quite different. XTRA produces no ambiguities in its parse tree. In each case slot (see Fillmore, 1978) underneath the verb-sense there are word senses rather than the word from the original sentence.

Our second parser involves a semantics-driven concept. Work is already under way on building a semantics driven natural language analyzer which addresses the well-known linguistics problems of language analysis. The justifications for doing this are the problems with previous efforts and the need for an adequate semantic analysis program. Ball and Wilks are currently working on an implementation of preference semantics using case grammar as a semantic base. The system is semantics driven because input sentences are analyzed to identify and correlate semantic chunks. Prominent semantic chunks are the action or state and the cases related to these. An object can be in some place at some time and a given action can take place in some location at some time. Semantics calls syntax to aid in the identification of semantic chunks. Say, some agent is expected by the semantics. Then a call is made to the syntactic component to see if the next element of the input can be a noun phrase. The semantic component expects certain constituents and uses syntax to verify such expectations.

6.9. Other work on meaning representations

There has been much work on building meaning representations of natural language utterances and we can not claim to do justice to all of those here. We shall begin with representations of natural language utterances on operating systems and then move on to more general approaches on meaning representation.

The theory of how to represent natural language queries in the Unix Consultant (see Wilensky et al., 1986) has evolved over a number of years. Initially, the system used a phrasal analyzer called PHRAN (see Arens, 1986; Wilensky et al., 1984) which read sentences in English and produced representations to decode their meanings. PHRAN contained a knowledge base of pattern-concept pairs where patterns were descriptions of literal utterances that had many different levels of abstraction. For example, *<person> <give> <person> <object>* is a phrasal pattern. Each pattern had an associated conceptual template which is a piece of meaning representation. For example, associated with the phrasal pattern *<nationality> restaurant* is a conceptual template denoting a restaurant that serves <nationality> type food.

PHRAN's use of patterns and concepts is similar to our use of action templates and case structures. However PHRAN is a general parser and not specifically geared towards operating systems. There was no theory of embedding to contend with our own. Therefore, although PHRAN was a good general mechanism for producing meaning representations of English it was not very efficient as a parser of queries about operating systems.

The latest Unix Consultant implementation (see Wilensky et al., 1986) involves a new parser called ALANA (Augmentable LANguage Analyzer) written by Cox (1986). ALANA is an extension of the PHRAN parser described above. Although ALANA is a more advanced parser than PHRAN there is no description of how the parser may handle multiple action queries. Any discussion of ALANA shows only how single action queries are handled. Again, there is no description of an alternative theory that competes with ours of embedding.

Douglass and Hegner (1982) used case frames in the front end for the Unix Computer Consultant (UCC) system. Case frames were templates representing the main action of a clause and the constituents of the action, such as the actor and recipient of the action. The case frames corresponded to logical operations in an operating system, and therefore formed the main link between English-language operating system concepts and the formal semantic definitions of specific UNIX commands. The problem with these case frames was that they were too far removed from natural language input to be useful and also there was no great theory of how to combine case frames together to formulate good meaning representations of complex queries.

The SINIX consultant involves a natural language interface which produces meaning representations of English sentences. Although the SINIX parser (see Kemke, 1986, Section 2.6.3) uses case structures to build up sentence case frames we find no description of a theory of how case structures may be combined efficiently.

Matthews and Pharr (1987) describe a system called USCSH (University of South Carolina SHell) which is an active intelligent assistance system for UNIX. Although the dictionary in USCSH contains grammatical information little semantic information is included, as yet. It is intended that a meaning-structure grammar (see Chafe, 1970) will be included at a later stage. Their approach to constructing meaning representations of natural language queries involves no discussion of the logical structure of discourse or temporal ordering of actions.

Although we have described four approaches to building meaning representations for queries about operating systems there has been much research on building meaning representations for natural language sentences in general. The IRUS (Information Retrieval Using the RUS parser) system uses a formal Meaning Representation Language (MRL) (see Bates et al., 1986). MRL has a formal declarative semantics that can be expressed in predicate calculus or procedural semantics (see Woods, 1981). There is no particular theory of how to embed sequences of actions here.

Fillmore (1968, 1977) discusses how natural language sentences can be understood using knowledge in a form of case structures. Case structures are frames into which verbs may be parsed. Fillmore concerns himself more with the syntax of verbs than their semantics. He says Fillmore (1968), “The preceding sections have contained an informal description of a syntactic model for language...” (p. 61). Different verbs may link to a number of different frames and he explains which verbs are constrained to which cases in which frames. Although Fillmore gives a good description of different verbs and their properties he does not concern himself with the semantic questions of verbs like *print* affecting objects like *files* or directories. He does not describe any theory of embedding where different structures for various verbs can be linked together. He is largely concerned with single action sentences. Fillmore helps us in defining properties of verbs but not how such verbs are integrated in an operating system consultant.

In Fillmore (1968) we have a discussion on anaphoric processes. Fillmore says, ‘‘Anaphoric processes are best understood from the point of view of an extended concept of sentence conjunction. That is, every language has ways of simplifying sentences connected by conjunctions or subjunctions, and the processes used under these conditions seem too be exactly the same as those used in sentences connected in discourse’’ (p. 56). Although this may be a good argument for understanding anaphoric sentences we believe that this heuristic is true of our theory of embedding. For example, the query ‘‘How do I list the files in paul.courses?’’ followed by, ‘‘How do I print them?’’ in a dialogue interpreter should give the same embedding as ‘‘How do I list paul.courses and then print the files?’’ The representation of the two separate sentences should be the same as if they were connected.

Schank (1975) has worked on a deep representation of natural language sentences called *conceptual dependency*. Schank intends a very deep representation because he wishes to have a language free form. His representation is similar to our deep case structures. Schank’s theory entails a reduction of all utterances to combinations of primitive *predicates* chosen from a set of twelve *actions* plus state and change of state, together with the primitive *causation*, and seven role relations or *conceptual cases*. Schank sets up case frames for primitive acts as opposed to Fillmore’s concentration on the surface verbs of English.

Wilks (1975a, 1975b, 1976, 1978a, 1978b) developed a natural language understanding program which parsed English text into deep meaning representations. Wilks' parser constructed a meaning representation made up of *templates*, having the basic form of *agent-action-object* which are integrated by the use of *paraplates* and *inference rules*. The *templates* are built up from *formulas* which represent individual word senses. In the discussion of meaning representations above there is no discussion of semantic formulas because information about such word senses would already be maintained in the parser that analyzes English input.

Our deep case structures are like Wilks' templates as they contain actions, objects and agents. Wilks' idea of building paraplates from templates parallels ours of building embedded action representations from case structures. However Wilks would have different templates for different clauses whereas we only have different templates for different verbs. Also Wilks talks of linking paraplates with cases, whereas we talk of linking case structures by embedding them inside each other to denote temporal relations. In other words, we are talking of using more pragmatic structures rather than semantic ones. Of course the semantic structures do exist in the parser that analyzes input. Another difference between our embedded action representations and Wilks' paraplates is that the EREPs are constructed on the fly whereas Wilks' paraplates already exist in the system.

Wilks (1976) makes an important point in that we should only put those cases into a formula that are necessary to specify the meaning of say a verb. For example a LOCATION is necessary to specify the meaning of living although it need not be necessary to specify the meaning of drinking. This is the heuristic that we use to define formulas for the meaning of words in the system. Notice that we do not fall into the trap of doing what Wilks (1976, p. 27) argues people like Fillmore, Schank, and the Generative Semanticists should not do. Wilks says that they are involved in

‘‘...displaying a full underlying structure directly **without the processes that reach it**.’’ He says, ‘‘I argued earlier that each of those three gave only a filled-in, or final, structure which in itself gives no hints as to **how you get there** [his emphasis]’’ (p. 27). In fact, Fillmore has developed a surface oriented view of case whereas Schank uses a deep case representation. Wilks uses a representation in between the two and that is the philosophy we have used in developing OSCON.

6.10. Embedded representations are useful

Embedded action representations are a precise means of formalizing meaning relations between UNIX actions. English queries involving interrelated actions can be understood effectively using these action representations. In particular, EREPs provide a framework for building domain-specific information about embedded commands. The most significant feature of EREPs is that because they maintain an implicit notion of time, or ordering of actions, there is no need to represent temporal orderings themselves. These are already inherently provided by the representation itself.

It is important in any natural language system which understands natural language queries about operating systems, that there be some mechanism for recognizing actions, and how they relate to other actions and objects. We believe that the above theory of embedded meaning representations for actions will be adequate in this endeavor. The inherent structure of the embedded action representations allows the system to build up a good temporal ordering of actions and objects. Wilks (1986) recognized that this ordering was important, ‘‘...our representation must have the ‘‘one after another’’ feature that texts have, rather than being static and timeless like most semantic nets..’’ (p. 10). The temporal ordering of actions is a more pragmatic characteristic of queries about operating systems that hasn’t been discussed much on other work in meaning representation.

There is much work yet to be done on EREPs. For example, we have not defined the rules for matching case structures to output from a parser, or promoting objects form one embedded action to another. There has been no discussion of the mechanisms involved in rejecting incorrect action relations occurring in user queries. This would happen if a user query did not match one of the action templates. For example a PRINT action could never be nested inside a DELETE action when they apply to the same file because if a file is deleted it is not possible to print the file. However, we need to investigate what OSCON should do when such errors are detected. Early detection of pragmatic user errors will increase the efficiency of the operating system consultant.

Chapter 7: The *OSCON* system

So far we have discussed a theoretical design of the natural language understander for an operating system consultant. We have not put great effort into describing the understander as a complete unit. Nor have we discussed how that understander relates to the knowledge base which solves or answers queries. This chapter deals with a general description of OSCON and the understander. We also describe the plan understander called *PlanCon*, which is a program that computes the rules of consequence and the rule of composition over Transfer Semantics action frames.

7.1. Design principles

We have taken the approach of building an operating system consultant which operates in real-time and which embodies a natural language understander. As was shown in Chapter 2, that happens to be a good approach to building any consultant system.

We have already shown that any good consultant system must provide a friendly interface to the user. The interface should not require the user to have any special computer skills, otherwise we defeat the purpose of the system. A natural language understander will be the best at functioning as a friendly interface as that is the language of the user. Queries can be posed in English and any subsequent dialogue would be in English including the system responses.

We are primarily interested in modeling the UNIX operating system although other operating systems are of interest. The design of OSCON is intended to be general enough to give help on many operating systems. That design has been motivated by lessons learned in building earlier operating consultants. These systems, called UCC and Yucca, were discussed in Chapter 2. We have included two major design principles in the design of the OSCON system: (1) the principle of understanding and solving; (2) the principle of a general approach to operating system consultation.

7.1.1. The principle of separating understanding and solving

There are two main functions that any consultant system must address. Those are the functions of *understanding* and *answering* user queries. The problem of understanding a query is different to the problem of solving or answering one. Queries like, ‘‘How do I delete a file?’’ make perfect sense to any computer user who may have no particular knowledge of operating systems. Deleting files is one of the most common functions that any computer user may need to perform. The knowledge that the *rm* command is used to perform this task is not necessary to understand it. However, general knowledge about files and the act of deleting them is necessary for solving.

One of the principle design features of our system is that the process of understanding a query is separate from that of solving a query. We call this, the *principle of separation of understanding and solving* and it has been reported in Hegner (1987). Problems related to understanding include the control of ambiguity. For example, in the query, ‘‘How do I print a file with pageheaders?’’ the file may already have pageheaders and that is different from the file getting pageheaders when it is printed. The understanding phase of the system involves determining that ambiguity exists and then resolving that ambiguity.

7.1.2. A general consultant

We intend that the system will have an general flavor. By this we mean if some user asks a query in the context of one operating system, OSCON will have the capability of answering the query in terms of another. For example, a user may be asking queries about UNIX and suddenly say, ‘‘How do I use ‘dir’ to find the creation date of all the files in my directory?’’ However, there is no *dir* command in UNIX although there is one in TOPS-20. Of course, the equivalent command for UNIX is *ls -l*. It is hoped that OSCON will answer user queries on many operating systems, although we are focusing on UNIX. Other computer operating systems of interest are VMS, VM/CMS, and DOS.

7.1.3. Representing principles as architecture

OSCON has a two-module architecture. One module, called the natural language understander, has the function of understanding and answering English queries. The second module, or knowledge base, is detailed and formal. It functions as the solving or answering module. The knowledge base is being constructed at the University of Vermont by Dr. Steve Hegner. Work on the knowledge base is discussed extensively in Douglass & Hegner (1982), Hegner & Douglass (1984) and Hegner (1987). Our architecture is similar to that found in many natural language interfaces to database systems (see Waltz, 1975, 1978; Hendrix et al., 1978; Martin et al., 1983; Wallace, 1985). In these systems the formal knowledge base and query language already exist, and the task is to add a natural language front end. In the operating system consultant, we are designing both modules to be efficient and tailored specifically for the domain of consulting on systems. The two-module architecture is one of the principle design features of OSCON. As pointed out by Hegner (see Hegner 1987, p. 1) the two-module architecture facilitates the important principle of separation of understanding and solving. The two modules are connected by a formal query language called OSquel. A good description of OSquel is given in Hegner & Douglass (1984).

We have already shown that any system which communicates information on some domain must possess good knowledge about that domain. A key design principle of our system is the construction of a detailed formal, knowledge base and retrieval facility. The knowledge base responds to complex and detailed technical queries concerning both static and dynamic information.

7.2. An overview of the consultant

There are two ways in which a help utility may be incorporated into a system. The utility may be designed as an integral component of the system. This approach may be applied to an existing system by rewriting components which are already supplied by the help utility. The Cousin interface by Hayes (1983) reflects this strategy. We hope to install the operating system consultant on new systems with very little effort and that it be visible only to those who wish to use it. The system is conceived entirely as an external utility which may be installed just like a new editor or compiler. We wish to ensure maximum portability and usability of the system.

The user interface is not intended to be elaborate by any means. Many users use standard video terminals and OSCON is designed with the intent of providing a reasonable interface to the UNIX system via such terminals. In particular, the communicator will be invoked by typing an appropriate command name to the processor, and then the query itself can be typed into the system as natural language text.

The system will be transportable to a wide variety of UNIX and UNIX-like systems. Common Lisp has been promoted as a standard for lisp programmers.

Common Lisp will soon become available on a large number of UNIX systems. Therefore, we are implementing OSCON in Common Lisp. As the knowledge base research program is not a part of this thesis, we will not discuss it here. However, a detailed description may be found in Hegner (1987).

The flow of control in OSCON is as follows: Initially, the user's natural language query is translated into a formal query in OSquel. This step resolves any ambiguity in the natural language query. When this step is completed, the request for information about the domain is sent to the knowledge base. The next step solves the formal query and natural language issues are not involved at this stage. The final stage involves translating the instantiated formal query into a natural language form suitable for presentation to the user.

7.3. An overview of the understander

The natural language understander parses English sentences into a formal query language called OSquel. Formal queries are represented in the form <P A Q U>. P and Q represent preconditions and postconditions for any action A. U represents the particular person or user performing A.

The understander can be considered in terms of two distinct phases: (1) formal query generation, and (2) answer production. The formal query generation phase involves four components. Each component produces a new level of meaning representation for some query. The need for having various levels of meaning representation in any interface is discussed by Sparck-Jones (1983). She tells us that in order to preserve text structure, and in order to do extensive inference, representations at different levels are required. She describes one current project of building a natural language front end to a database where different meaning representations must be utilized.

The control flow of the understander proceeds like this: (1) Initially, an English query is input by the user. The query is parsed into a *shallow representation* by a natural language parser. The term *shallow representation* is used to describe the output from different parsing techniques. This representation may include some semantics such as knowledge of word senses. Examples of natural language parsers we currently use are described by Ball and Huang in Wilks (1986) and by Slator in Wilks et al. (1987); (2) Each shallow representation is passed to an embedded action representation (EREP) generator. This component builds semantic representations of queries from the shallow representation and makes use of semantic case frames existing in a database. Case labels are attached to various items; (3) Each embedded action representation is passed to a Transfer Semantics component which maintains a database of knowledge frames. The Transfer Semantics component is the heart of the understander. It contains the abstract knowledge about operating systems and embodies the tasks of frame selection and instantiation; (4) A domain-specific Transfer Semantics representation is passed to a formal query generator which produces an uninstantiated formal query to the database in a language called OSquel. Formal queries are instantiated by the application of a solving process in the knowledge base. The answer generation phase of the understander is concerned with producing natural language answers from instantiated queries.

7.4. The PlanCon program

The function of PlanCon is to compute inference rules over Transfer Semantics. Transfer Semantics was described in Chapter 3 and the rules were described in Chapters 4 and 5. Transfer Semantics on its own is not powerful enough for understanding more complex queries and that is why PlanCon is used. Presently, the first and second rules of consequence and the general rule of composition (ROC_0) have been implemented. We have implemented the rules of consequence over the PRINT frame, and the rule of composition for the LIST and PRINT frames.

All frames are to be loaded into the system before any computation begins. Therefore, object and action frames are input by the programmer. We show the precondition set for the PRINT frame in Figure 7.1.

```
(preconditions

   (mandatory (not (o-frame directory-file)))

   (optional (((o-frame file)
               (has (o-frame contents)
                    (has (o-frame visible-byte-sequences)))))

              (((o-frame file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences))))
                (o-frame print-queue))

              (((o-frame file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences)))))))
```

Figure 7.1 Precondition set for PRINT.

The first condition in the precondition set is mandatory. The next three conditions are optional. The final optional condition is a default. Interpreting the above set, it is noted that the mandatory condition specifies that a directory file should not be printed. Now, let us not confuse the reader at this point. Of course, it is possible to print a directory by first listing it and then printing it. Yet, one does not print directories themselves, and this is what we are concerned with here. The first optional condition specifies a preference that files are printed and their contents are preferably visible byte sequences. The second optional condition declares in addition the existence of a printer queue. In order to print a file on the printer it is certainly useful to have a printer queue. Finally, the third precondition in the set is a default, and is the same as the first optional condition. We do not worry about preconditions such as the system being up, the terminal working or keyboard on-line. These are simply assumed. The postcondition set for PRINT is shown in Figure 7.2.

```
(postconditions

   (optional  (((o-frame non-directory-file)
                 (has (o-frame contents)
                      (has (o-frame visible-byte-sequences)
                           (has (o-frame filter)))))
                 (o-frame device-file))

              (((o-frame non-directory-file)
                 (has (o-frame contents)
                      (has (o-frame visible-byte-sequences)
                           (has (o-frame filter))))))

             (((o-frame non-directory-file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences))))
              (o-frame device-file))))
```

Figure 7.2 Postcondition set for PRINT.

There are three optional conditions, the final one delimiting a default. The first condition declares that the file which we saw in the precondition set also exists in the postcondition set. The file doesn't disappear after printing as would be the case with a delete frame. The file still contains visible-byte-sequences although a filter is now also applied. Filters are items such as pageheaders, line numbers and dates. Also a device file exists to denote default standard output which is the terminal screen.

The second optional condition tells us that a print queue exists and has a print queue entry. Also, a filter may be applied to the contents of the file. The third post-condition in the set is again a default and specifies output to a device file.

For each frame there must be some commands that can execute the action. The commands for printing are shown in Figure 7.3.

```
(actions
   (optional (o-frame cat)
             (o-frame more)
             (o-frame lpr)
             (o-frame pr)
             (o-frame print)
             (o-frame option-list)))
```

Figure 7.3 Action set for PRINT.

Printing can be completed with any of the commands in the optional set of actions and their respective options. Finally, in Figure 7.4 we specify the actor performing the action or transfer. Any user can print a file and this is represented in the actor set.

In the preceding examples we make no claim that the action frame components are in any way complete or sufficient in order to describe the action of printing. Indeed we expect to extend the precondition and postcondition sets to handle more complex queries.

```
(actor
   (optional (o-frame user)))
```

Figure 7.4 Actor for PRINT.

7.4.1. Computing the first rule of consequence

Say the user asks, ‘‘How do I print a file on the screen?’’. A problem with this query is that the frame matcher cannot match *file* in the query to *non-directory-file* in the postcondition set for the PRINT action frame. The postconditions in the frame are too specific. All PlanCon needs to do is to run the first rule of consequence over the postconditions for the print frame.

After applying the first rule of consequence we get Figure 7.5, which shows files rather than non-directory-files. The frame matcher will match file from the query to file in the postcondition set now. The problem of strong postconditions is solved.

7.4.2. Computing the second rule of consequence

Say, the user has entered the query, ‘‘How do I print a plain file?’’ Assume that the print frame has been selected from a number of frames as the candidate that we should use. The problem here is that plain file is not mentioned in the precondition set. The frame matcher would not like the print frame at all. However by applying the first rule of consequence to Figure 7.1 we can derive Figure 7.6.

```
(postconditions

    (optional  (((o-frame file)
                  (has (o-frame contents)
                       (has (o-frame visible-byte-sequences)
                            (has (o-frame filter)))))
                  (o-frame file))

               (((o-frame file)
                  (has (o-frame contents)
                       (has (o-frame visible-byte-sequences)
                            (has (o-frame filter))))))

             (((o-frame file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences))))
               (o-frame file))))
```

Figure 7.5 Weakening the postconditions for PRINT.

In Figure 7.6 we show the strengthened preconditions for printing. Now the frame matcher has no problem in matching “How do I print a plain file?” to the first optional in the set.

7.4.3. Computing the rule of composition

The rule of composition defined that certain frames could be concatenated together in sequence. For example, the query, “How do I list a directory and then print it?” could be handled by concatenating the LIST and PRINT frames. The preconditions for LIST are the complete preconditions for the sequence. In Figure 7.7 we show the optional preconditions from the set for LIST, particular to the above query.

```
(preconditions

   (mandatory (not (o-frame directory-file)))

   (optional (((o-frame plain-file)
               (has (o-frame contents)
                    (has (o-frame visible-byte-sequences)))))

             (((o-frame plain-file)
               (has (o-frame contents)
                    (has (o-frame visible-byte-sequences))))
               (o-frame print-queue))

             (((o-frame plain-file)
               (has (o-frame contents)
                    (has (o-frame visible-byte-sequences))))))
```

Figure 7.6 Strengthening the preconditions for PRINT.

```
(preconditions

   (optional (((o-frame file)
               (has (o-frame contents)
                    (has (o-frame visible-byte-sequences))))))
```

Figure 7.7 Selected preconditions for LIST.

The correct optional postcondition for LIST is shown in Figure 7.8. The selected postcondition represents the output of listing directories coming out on the screen. The next step is to apply this as a precondition to the PRINT frame. These preconditions for PRINT produce the postcondition shown in Figure 7.9. The rule of composition for listing and printing has now been completed. Of course, this computation has involved determining what the PRINT frame should have if given the precondition set in Figure 7.7.

```
(postconditions

   (optional  (((o-frame file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences))))
                (o-frame device-file))))
```

Figure 7.8 Selected postconditions for LIST.

```
(postconditions

    (optional (((o-frame non-directory-file)
                (has (o-frame contents)
                     (has (o-frame visible-byte-sequences))))
               (o-frame device file))))
```

Figure 7.9 Selected postconditions for PRINT.

Chapter 8: Conclusion

A theory of understanding queries about computer operating systems has now been presented and it is time to summarize what has been done and more important, what we have to do. There are many areas of the problem not covered by the thesis, while other areas have been discussed adequately. We present first thoughts on ideas for further work and some problems with our existing framework.

8.1. Summary

The thesis has described a theoretical model of a natural language understander for an operating system consultant. The first step in doing this was to describe a good knowledge representation for operating systems. The next step was to define a set of inference rules which could operate on that representation and derive new information. It was shown that without inference rules natural language queries could not be understood. The construction of inference rules involved defining a language for describing the rules. The language was borrowed from Axiomatic Semantics. From there, the power of a plan understander, called *PlanCon*, was extended by introducing new rules of inference. These new rules extended the rule of composition in various directions, and covered selective, parallel, and fork composition. We demonstrated that the general rule of composition had really defined the inherent properties of composition by showing cases where each of the other composition rules collapsed into the general one.

Then we changed the direction of discussion. Instead of working out the knowledge representation to meet natural language queries (KNOWLEDGE => LANGUAGE) we worked out a representation of language to meet knowledge

(LANGUAGE => KNOWLEDGE). We showed how a theory of meaning representations for natural language queries could be developed and how that theory would represent examples of natural language queries. The theory of the understander was placed in the context of a complete description of OSCON and we described how PlanCon exercised three of the inference rules over Transfer Semantics.

8.2. Consultation by artificial communication is useful

In developing the work herein we conclude that a natural language operating system consultant will be useful to those people who wish to use one. We have not only shown why an artificial consultant would be useful, but have also developed a theory of an understander which could be incorporated within one. The implemented system will take natural language input and parse it into some meaning representation and that can be further translated into a detailed domain specific representation. We conclude that the knowledge representation and planning components of any consultant system are a major part of its final success. That is why we concentrated so heavily upon them in Chapters 4, 5, and 6.

Detailed representations of the input queries can be passed to a knowledge base where answers to queries can be produced. We believe that if the knowledge representation is extensive enough, and the natural language parser circumscribes a large part of English query formation, then the consultant would be a good approximation of what human consultants do. Of course, the program may be slow, and a good indicator of that is the extent of the theory in this thesis. And remember, we have only described the understander here. However, we do not worry about such slowness. In the future, computer architectural developments and hardware implementations will increase the speed of computers dramatically. Also, using a artificial consultant would certainly be better than looking through a large set of manual pages or using a simple key-on-command system, or even worse by asking a computer expert. Computer experts can be the worst consultants of all.

8.3. Towards better consultants

Although we have developed a theory of an understander for queries about operating systems, there are many ideas that have not been explored. In order to build better consultants these areas must also be investigated. Let’s talk about some of the things that we haven’t done yet.

One of the major components that our theory lacks is a formulation of answer generation phase for OSCON. ‘‘What happens when the query solver has formulated an answer to a query?’’, ‘‘How do we express that answer in English?’’ Of course, the major aim of the thesis was not to do this and it is certainly an area of further investigation. We postulate that Transfer Semantics will be useful in presenting answers to the user. It is suspected that answer generation will not be too difficult as the query solver produces answers in the form <P A Q U>. The answering process may involve an inversion of the meaning representation approach in conjunction with Transfer Semantics.

There was no development of a theory of how natural language sentences could be parsed into syntactico-semantic structures. However, this is a large area of research and much work has already been done there. As was already mentioned in Chapter 7 we would hope to use existing parsers as front ends to do this.

There has been no in-depth discussion of the frame selection mechanism or how knowledge is selected from meaning representations. We described briefly that frames would be selected on a preference basis where the frame with the most satisfied over non-satisfied matches is the right one. However there are better detailed ways of selecting frames rather than simple preference. This is an area of future study.

8.4. Detection of user misconceptions

A good consultant system must have some way of deciding and reporting that the user is making mistakes while using the program. For example, say a user asks the query, “How do I print a file with the -Z option?”. “-Z” is not an option on printing. Nor, can -Z be inferred for printing. The action frame for *printing* does not specify a formula of the form ({P} A {Q}) because A is not satisfied. The user could be informed that he is trying to do something impossible. We also hope to investigate the possibility of recognizing ill-formed embedding. For example, the query, “How do I delete my files and then list them?” doesn’t make much sense at all. Files cannot be listed after they are deleted. Error checking components could be added to the system where errors would be detected at early stages and therefore not passed on to the knowledge base. The knowledge base would find out that the queries were incorrect but that would happen a lot later, and therefore is less efficient. It is envisaged that PlanCon will be able to detect errors from implicit plans existing in user queries.

8.5. The necessity of understanding context

One major problem which must be tackled is the understanding of user queries in context. For example, a user may ask, ‘‘How do I print the file paul on the Imagen?’’ and then say, after a few more sentences, ‘‘What happened with paul?’’. There must be a mechanism for sorting temporal representations of sentences on a stack and pulling them off later when they are needed. This is where our theory of embedded representations comes in. We do not think of contextual understanding in terms of single sentences but embedded representations of actions. Two or more queries would produce a concatenated representation of two or more sentences. Of course, we could not store representations every time someone asks a question. There would be too many stored and this would violate our minimal storage hypothesis. There needs to be some theory of how to forget information in the EREPs as time goes on.

A useful element of any embedded temporal representation is evidence of *focus* (see Carberry, 1985). For example, if someone is asking a lot of questions about printing, then many of the action frames on the embedded stack will involve PRINT frames. That is useful to know because it will aid in understanding and frame selection. If a user seems to be asking queries about printing, and OSCON recognizes that, then it will help in understanding subsequent queries. Also if two frames have the same number of satisfied preferences, it would be useful to know about focus as OSCON could then take a better shot at selecting the correct frame.

8.6. The representation of belief

One major area of further investigation is the representation of belief in the system. Belief models are useful as an aid for understanding why the user makes mistakes in a dialogue system. The emphasis in this area is to consider situations where belief structures of the user differ radically from the beliefs and plans of OSCON. This is useful for the modeling of both cooperative and adverse situations where a system must have a model of what its interlocuter does and does not know.

Good discussions of belief models are found in Wilks & Ballim (1987) and Ballim (1986). Barnden (1986) describes an alternative method of representing belief. We intend to include ideas about belief representation from Wilks & Ballim's and Barnden's models in OSCON. Ballim (1986) proposes a model for computing the beliefs of others on a default basis with emphasis on the notion of self-knowledge. A belief model will be particularly useful in developing representations of user knowledge and determining whether a user is naive or an expert. The answer generation mechanism would be able to present answers to the user depending on whether the user has a detailed knowledge of operating systems or not. Any belief generator allows a system to develop a good representation of what the user believes about the system. The belief system will be useful in determining when the user is misinterpreting the system by believing that it is communicating about one system (say, TOPS-20) while it is really communicating about another (say, UNIX).

One of the major functions of the understander is to decide the intent of the user. The intent of a user is determined after frame selection and instantiation occur. When some frame has been instantiated, there will be a primary unknown within that frame. For example, with the query, ‘‘How do I print a file on the laser printer?’’ the action component (A) in the frame for printing will not be instantiated. Therefore, OSCON marks the action component of the frame as being the information required. In the case of a query like, ‘‘What happens if I delete all my files?’’ the precondition is matched, yet the postcondition is not specified and is therefore the information that the user requires. Belief models will also help to determine the intent of the user.

The above ideas are currently under investigation and we hope to report on them soon. All of these will help toward building better consultant systems. We will close with a question raised by Wilensky et al. (1984, p. 590) , ‘‘Probably the most significant problem in UC (Unix Consultant) involves representational issues. That is, how can the various entities, actions and relationships that constitute the UC domain best be denoted in a formal language?’’ It is hoped that this thesis makes a start in answering this question and many more.

References

Algaić, S. and M.A. Arbib (1978) *The design of well-structured and correct programs*. New York: Springer-Verlag.

Alterman, Richard (1986) *An adaptive planner*. In Proc. Fifth National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, Vol. 1 (Science), pp. 65-69, August.

Arens, Yigal (1986) *CLUSTER: An approach to contextual language understanding.* Report No. UCB/CSD 86/293, Computer Science Division (EECS), University of California, Berkeley, California 94720, April.

Ballim, Afzal (1986) *The computer generation of nested points of view*. Master's thesis, Computer Science Department, Dept. 3CU, Box 30001, New Mexico State University, Las Cruces, New Mexico, NM 88003-0001.

Barnden, John (1986) *A viewpoint distinction in the representation of propositional attitudes*. In Proc. Fifth National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, Vol. 1, pp. 411-415, August.

Bates, Madeleine, M.G. Moser & David Stallard (1986) *The IRUS transportable natural language database interface*. Expert Database Systems, In Proc. First International Workshop, Larry Kerschberg (Ed.), pp. 617-630, Benjamin/Cummings Publishing Company, Inc.

Billmers, Meyer A. & Michael G. Garifio (1985) *Building knowledge-based operating system consultants*. In Proceedings of the Second Conference on Artificial Intelligence Applications, pp. 449-454, Miami Beach, Florida, December.

Bobrow, D.G. & T. Winograd (1977) *An overview of KRL, a knowledge representation language*. Cognitive Science, Vol. 1, No. 1, pp. 3-46.

Bougarev, Bran (1979) *Automatic Resolution of Linguistic Ambiguities*. Technical Report No. 11, University of Cambridge Computer Laboratory, Cambridge, United Kingdom.

Brachman, R.J. (1979) *On the epistemological status of semantic networks*. In Associative Networks: Representation and use of knowledge by computers, pp. 3-50, N.V. Findler (Ed.). New York: Academic Press.

Carberry, Sandra (1983) *Tracking user goals in an information-seeking environment*. In Proc. Second National Conference on Artificial Intelligence (AAAI-83), pp. 59-63, University of Maryland and George Washington University, Washington, DC, August.

________ (1985) *A pragmatics-based approach to understanding intersentential ellipsis*. In Proc. 23rd Annual Conference of the Association for Computational Linguistics, pp. 188-197, Chicago, Illinois, July.

Chafe, W.L. (1970) *Meaning and structure of language.* Chicago, Illinois: University of Chicago Press.

Cox, Charles A. (1986) *ALANA Augmentable LANguage Analyzer*. Report No. UCB/CSD 86/283, Computer Science Division (EECS), University of California, Berkeley, California 94720, January.

Dearholt, D.W.; R.W. Schvaneveldt & F.T. Durso (1985) *Properties of networks derived from proximities*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-85-14, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Douglass, Robert J. & Stephen J. Hegner (1982) *An expert consultant for the UNIX operating system: Bridging the gap between the user and command language semantics*. In Proc. Fourth National Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)/SCIEO Conference, pp. 119-127, Saskatoon, Saskatchewan, May.

Fass, D.C. (1986a) *Collative Semantics: a description of the Meta5 program*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-23, Computing Research Laboratory, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

________ (1986b) *Collative Semantics: an approach to coherence*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-56, Rio Grande Research Corridor, Computing Research Laboratory, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Fass, D.C. & Yorick Wilks (1983) *Preference semantics, ill-formedness and metaphor*. American Journal of Computational Linguistics, Vol. 9, pp. 178-187.

Fikes, R.E. & N.J. Nilsson (1971) *STRIPS: A new approach to the application of theorem proving to problem solving*. Artificial Intelligence, Vol. 2, pp. 189-208.

Fillmore, C.J. (1968) *The case for case*. In Universals in Linguistic Theory, E. Bach and R. Harms (Eds.), pp. 1-90. New York: Holt, Rinehart and Winston.

________ (1977) *The case for case reopened*. In Syntax and Semantics, Peter Cole and Jerrold M. Sadock (Eds.), pp. 59-81. New York: Academic Press.

Floyd, R.W. (1967) *Assigning meanings to programs*. In Mathematical Aspects of Computer Science, Proc. American Mathematical Society, Symposium in Applied Mathematics, Vol. 19, J. T. Schwartz (Ed.), Providence, Rhode Island, pp. 19-31.

Gabrielan, A. & M.E. Stickney (1987) *Hierarchical representation of causal knowledge*. Proc. Western Conference on Expert Systems (WESTEX-87), pp. 82-89, Disneyland Hotel, Anaheim, California, July.

Goldstein, I.P., & R.B. Roberts (1977) *Nudge, a knowledge-based scheduling program*. In Proc. Fifth International Joint Conference on Artificial Intelligence (IJCAI-77), pp. 257-263, Cambridge, Mass.

Hayes, Philip J. (1982) *Uniform help facilities for a cooperative user interface*. In Proc. National Computer Conference, pp. 469-474, Houston, Texas.

Hayes, Philip J. & Pedro A. Szekely (1983) *Graceful interaction through the COUSIN command interface*. International Journal of Man-Machine Studies, Vol. 19, pp. 285-306.

Hegner, Stephen J. (1987) *Representation of command language behavior for an operating system expert consultation facility*. Technical Report CS/TR87-02, CS/EE Department, University of Vermont, Burlington, Vermont, USA.

Hegner, Stephen J. & Robert J. Douglass (1984) *Knowledge base design for an operating system expert consultant*. In Proc. of the Fifth National Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI), pp. 159-161, London, Ontario, December.

Hendrix, G.G., E.D. Sacerdoti, D. Sagalowicz & J. Slocum (1978) *Developing a natural language interface to complex data*. ACM Transactions on Database Systems (TODS), Vol. 3, No. 2, pp. 105-147, June.

Hoare, C. A. R. (1969) *An axiomatic basis for computer programming*. Communications of the ACM, Vol. 12, No. 10, pp. 576-583, October.

Hoare, C.A.R. & N. Wirth (1973) *An axiomatic definition of the programming language PASCAL*. Acta Informatica, Vol. 2, pp. 335-355.

Huang, Xiuming (1985) *Machine translation in the semantic definite clause grammars formalism*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-72, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Kautz, Henry A. & James F. Allen (1986) *Generalized plan recognition*. In Proc. Fifth National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, Vol. 1 (Science), pp. 32-37, August.

Kemke, Christal (1986) *The SINIX Consultant — Requirements, Design, and Implementation of an intelligent Help System for a UNIX Derivative*. Universitat des Saarlandes, KI-Labor (SC-Project), Bericht Nr. 11, October.

________ (1987) *Representation of domain knowledge in an intelligent help system*. In Human-Computer Interaction — INTERACT ’87, H.J. Bullinger and B. Shakel (Eds.), pp. 215-220. Amsterdam: Elsevier Science Publications B.V. (North-Holland).

Litman, Diane J. & James F. Allen (1984) *A plan recognition model for clarification subdialogues*. In Proc. Tenth International Conference on Computational Linguistics, and 22nd Annual meeting of the Association for Computational Linguistics (COLING-84), pp. 302-311, Stanford, California, July.

Martin, Paul, Douglas Appelt & Fernando Pereira (1983) *Transportability and generality in a natural-language interface system*. In Proc. Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), pp. 573-581, Alan Bundy (Ed.), Karlsruhe, West Germany, August.

Matthews, Manton & Walter Pharr (1987) *Knowledge acquisition for active assistance.* Preprints of the First International Workshop on Knowledge representation in the UNIX help domain, University of California, Berkeley, California, December.

McDonald, James E., J.D. Stone & L.S. Liebelt (1983) *Searching for items in menus: The effects of organization and type of target*. In Proceedings of the 27th Annual Meeting of the Human Factors Society, pp. 834-837, Norfolk, Santa Monica, October.

McDonald, James E., Donald W. Dearholt, Kenneth R. Paap & Roger W. Schvaneveldt (1986) *Human factors in computing systems.* Proc. CHI’86 conference, Special issue of the SIGCHI Bulletin, pp. 285-290, Marilyn Mantei & Peter Orbeton (Eds.), Boston, Mass., April.

McDonald, James E. & Roger W. Schvaneveldt (1987) *The application of user knowledge to interface design*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-87-93, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul (1987) *Natural language interfaces in computer aided instruction — What happened before and after the 80s AICAI coup*. In Proc. Fourth International Symposium on Modeling and Simulation Methodology, University of Arizona, Tucson, Arizona, January.

Minsky, Marvin (1975) *A framework for representing knowledge*. In The psychology of computer vision, P.H. Winston (Ed.), New York: McGraw-Hill.

Owicki, S. & D. Gries (1976a) *An axiomatic proof technique for parallel programs I*. Acta Informatica, Vol. 6, pp. 319-340.

________ (1976b) *Verifying properties of parallel programs: an axiomatic approach*. Communications of the ACM, Vol. 19, pp. 279-285.

Pagan, Frank G. (1981) *Formal specification of programming languages: A panoramic primer*. Englewood Cliffs, New Jersey: Prentice-Hall.

Pelavin, Richard N. & James F. Allen (1987) *A model of concurrent actions having temporal extent*. In Proc. Sixth National Conference on Artificial Intelligence (AAAI-87), pp. 246-250, Seattle, Washington, Vol. 1, July.

Pereira, Fernando C.N. & D. H. D. Warren (1980) *Definite clause grammars for language analysis — A survey of the formalism and a comparison with augmented transition networks*. Artificial Intelligence, Vol. 13, No. 3, pp. 231-278, May.

Pollack, Martha E. (1986) *A model of plan inference that distinguishes between the beliefs of actors and observers*. In Proc. of the 24th Annual Meeting of the Association for Computational Linguistics (ACL) Conference, pp. 207-214, Columbia University, New York, New York, June.

Sandewall, Erik & Ralph Ronnquist (1986) *A representation of action structures*. In Proc. Sixth National Conference on Artificial Intelligence (AAAI-87), Seattle, Washington, Vol. 1, pp. 89-97, July.

Schank, R.C. (1975) *Conceptual information processing*. Amsterdam: North-Holland.

Schank, R.C. & R.P. Abelson (1977) *Scripts, plans, goals and understanding: an enquiry into human knowledge structures*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Sparck-Jones, Karen (1983) *Shifting meaning representations*. In Proc. Eighth International Joint Conference on Artificial Intelligence (IJCAI-8), Alan Bundy (Ed.), pp. 573-581, Karlsruhe, West Germany, August.

Stanat, Donald F. & David F. McAllister (1977) *Discrete mathematics in computer science*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc.

Tyler, Sherman W. & Siegfried Treu (1986) *Adaptive interface design: a symmetric model and a knowledge-based implementation*. The third ACM-SIGOIS conference on Office Information Systems, Association of Computing Machinery, SIGOIS Bulletin (formerly SIGOA Bulletin), Vol. 7, Nos. 2-3, pp. 53-60, Summer-Fall.

Wallace, Mark (1985) *Communicating with databases in natural language*. Chichester, England: Ellis Horwood Limited.

Waltz, David (1975) *Natural language access to a large database: an engineering approach*. Advance papers for the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75), pp. 868-872, Tbilisi, Georgia, USSR, September.

Waltz, David (1978) *An English language question answering system for a large relational database*. Communications of the ACM, Vol. 21, No. 7, pp. 526-539, July.

Wilensky, Robert (1982) *Talking to UNIX in English: An overview of an On-Line Consultant*. Report No. UCB/CSD 82/104, Computer Science Division (EECS), University of California, Berkeley, California 94720, September.

________ (1986) *Some problems and proposals for knowledge representation*. Report No. UCB/CSD 86/294, Computer Science Division (EECS), University of California, Berkeley, California 94720, May.

________ (1987) *Some complexities of goal analysis*. Preprints of the Third Conference on Theoretical Issues in Natural Language Processing (TINLAP-3), Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, pp. 97-99, January.

Wilensky, Robert, Yigal Arens & David Chin (1984) *Talking to UNIX in English: An overview of UC*. Communications of the ACM, Vol. 27, No. 6, pp. 574-593, June.

Wilensky, Robert, Jim Mayfield, Anthony Albert, David Chin, Charles Cox, Marc Luria, James Martin and Dekai Wu (1986) *UC — a progress report*. Report No. UCB/CSD 87/303, Computer Science Division (EECS), University of California, Berkeley, California 94720, July.

Wilks, Yorick (1975a) *An intelligent analyser and understander of English*. Communications of the Association of Computing Machinery (ACM), Vol. 18, pp. 264-274.

________ (1975b) *A preferential, pattern-seeking, semantics for natural language inference*. Artificial Intelligence, Vol. 6, No. 1, pp. 53-74.

________ (1976) *Processing case*. Technical Report, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland. Also in American Journal of Computational Linguistics, microfiche 56.

________ (1978a) *Good and bad arguments about semantic primitives*. Communication and Cognition, Vol 10., No. 3/4, pp. 181-221.

________ (1978b) *Making preferences more active*. Artificial intelligence, Vol. 11, pp. 197-223.

________ (1986) *Projects at CRL in Natural Language Processing*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-58, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Wilks, Yorick; Xiuming Huang and Dan Fass (1985) *Syntax, Semantics and Right Attachment*. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85), pp. 779-784, Los Angeles, California.

Wilks, Yorick & Afzal Ballim (1987) *Multiple Agents and the Heuristic Ascription of Belief*. In Proc. of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Vol. 1, pp. 118-124, Milan, Italy.

Wilks, Yorick, Dan Fass, Cheng-Ming Guo, James E. McDonald, Tony Plate & Brian M. Slator (1987) *A tractable machine dictionary as a resource for computational semantics*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-87-105, Computing Research Laboratory, Dept. 3CRL, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Woods, W.A. (1981) *Procedural semantics as a theory of meaning*. In Elements of Discourse Understanding, A. Joshi, B.L. Webber and I. Sag (Eds.). Cambridge, Mass.: Cambridge University Press.

Yun, David Y. Y. & D. Loeb (1984) *The CMS-HELP expert system*. In Proc. of the International Conference on Data Engineering, IEEE Computer Society, pp. 459-466, Los Angeles, California.