Understanding complex queries on operating systems †

Paul Mc Kevitt

Computing Research Laboratory
Dept. 3CRL, Box 30001
New Mexico State University
Las Cruces, NM 88003-0001, USA.
CSNET: paul@nmsu.edu [(505) 646-5942/5466]

Submission: First Irish National Conference on AI
Suggested Topic of Interest: Knowledge Representation/
Applications of Knowledge bases/ Natural language processing
Keywords: natural language queries, knowledge representation, operating systems.
Paper Length: 4989 words

ABSTRACT

When building knowledge representation schemes for particular domains we should realize abstractions of relationships from those domains. Computer operating systems involve numerous actions or commands which can transfer data from one state to another. It is this process of transfer that should be formalized in any representation of such actions. Transfer Semantics already exists as a knowledge representation scheme for operating system commands (Mc Kevitt & Wilks, 1987). Yet, no such scheme is powerful without inferencing. Axiomatic semantic techniques have been applied in exploring the logical foundations of computer programming. We can borrow axiomatic semantics as a language for constructing abstract formalizations of inference rules for Transfer Semantics. In particular, complex commands in operating systems such as UNIX∗ can be formalized by this method.

†This research is currently funded by
U S *WEST* Advanced Technologies
under their Sponsored Research Program.

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

0. Introduction

A knowledge representation scheme is never complete while there are no strategies to manipulate that scheme. We have developed a knowledge representation for computer operating systems called Transfer Semantics (see Mc Kevitt & Wilks, 1987). Transfer Semantics is a powerful abstract semantics used in understanding natural language queries about operating systems. The semantics is abstract because a detailed knowledge of operating systems is not necessary to understand user queries. It is the process of providing answers for queries that requires specific detailed knowledge. The work of answering natural language queries is performed by a formal knowledge base.

Transfer Semantics is a knowledge representation used to formalize operating system actions and any objects affected by such actions. We use Transfer Semantics to represent the means by which operating system commands transfer objects from one state to another. In Transfer Semantics operating system objects are represented by object frames. The object frames are structured in a tree-like representation. Action frames are used to specify transfer relations among object frames.

Each action frame is a formal representation of operating system actions or commands. Action frames consist of preconditions, postconditions, actions and actors. Preconditions are sets of states of objects existing before commands take effect. Postconditions involve sets of states of objects after a command is performed. Such conditions specifying states of objects are ‘preferred’, i.e. we do not specify all conditions on frames, only those that usually occur. Actions include the particular command(s) that cause(s) any transfer of object states. An actor is any person capable of performing some action. Preconditions and postconditions are mentioned in most of the literature on planning and have been used for specifying plans and goals. For example, 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.

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. What we mean by weak and strong will become apparent later on. 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 by defining and applying various rules of inference. Inference rules will be used to manipulate the preferred conditions in each frame so that the scope of frame meaning can be expanded. This paper is about: (1) defining some rules of inference, (2) how these rules can be used to strengthen Transfer Semantics, and more important, (3) why we designed a weak semantics in the first place.

We begin in section one with a brief overview of the Operating System CONsultant (OSCON) system and its relation to other work. We follow in section two with examples of natural language queries that demonstrate problems in Transfer Semantics. Section three describes a language for representing inference rules. In section four some general rules of inference are defined to solve the problems. This section includes a number of worked examples on how those inference rules may be applied. A description of other work relating to our inferencing schema comes next. As always, we arrive at some conclusions, and in section six the need for inference processes is justified.

1. The Operating System Consultant

At the Computing Research Laboratory we are developing a natural language understander for a consultant system called OSCON. OSCON is being programmed in Kyoto Common Lisp. The system is intended to help novice and expert users learn operating system concepts. 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 TOPS-20*, VMS* and VM/CMS†.

The system will have an ‘interdisciplinary’ 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 our intention to build a system that will allow the user to specify which operating system he wishes to learn. It would be possible for the system to answer a query about say, ‘‘deleting’’ by showing the necessary commands for different operating systems or just a specified one.

OSCON has a two-module architecture involving a natural language understander and a knowledge base. The natural language understander has the function of understanding and answering English queries. Query responses will be in English too. The knowledge base is a detailed, formal knowledge base and functions as a 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; Martin et al. 1983; Wallace 1985; & Hendrix et al. 1978). 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 an important principle of separation of understanding and solving.

In the natural language understander parsed English sentences are translated into a formal query language called OSquel (see Hegner, 1987). Formal queries are instantiated by the knowledge base and returned to the understander where answers are produced in English. 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 natural language 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: (1) shallow parser, (2) deep parser, (3) knowledge representation, and (4) formal query generator. Each component produces a new level of meaning representation for some query. The advantage of having various levels of meaning

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

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

representation in any interface is discussed by Sparck-Jones (1983).

The control flow of the natural language understander proceeds like this: (1) Initially, an English query is input by the user. The query is parsed into a ‘shallow representation’ by some natural language parser. This representation may include some semantics such as knowledge of word senses. Examples of natural language parsers we currently use are described by Ball & Huang in Wilks (1986) and by Slator in Wilks et al. (1987); (2) Each shallow representation is passed to an Embedded Concept Representation 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 (see Mc Kevitt, 1986a); (3) Each embedded concept representation is passed to a Transfer Semantics component which maintains a database of knowledge frames. The Transfer Semantics component is the heart of the natural language understander. It contains the abstract knowledge about operating systems and embodies the tasks of frame selection and instantiation (see Mc Kevitt, 1986b); (4) A domain-specific Transfer Semantics representation is passed to a formal query generator which produces an uninstantiated formal query to the knowledge base in the query 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.

1.1. Relation to other Operating System Consultants

Many researchers are working on building operating system consultants. There are three types of system being built: (1) systems with menu-based interfaces (MENU), (2) systems with limited natural language capability (LNL), and (3) systems with natural language interfaces (NL). In theory MENU, LNL, and NL systems could involve a command-level interface, i.e. the user could execute commands at the interface. However, many of these systems only act as consultants.

MENU systems are based on menu-selection and some of them include a command-level option, others do not. The problem with most menu-selection approaches for operating system consultants (pointed out by McDonald & Schvaneveldt 1987, p. 14; Hegner 1987, p. 1 and Wilensky et al. 1984, p. 576) 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. 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 extreme examples of MENU systems containing 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. The next four paragraphs discuss examples of MENU systems and that alone distinguishes our work from theirs.

(I) 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 & 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 for UNIX 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. We are working closely with the Cognitive Systems Group to provide empirical backing for any assumptions made in developing the natural language understander for OSCON.

(II) Tyler (1986) describes 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.

(III) 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. The COUSIN system provides two types of user friendly information: (1) static descriptions of possibly invoked subsystems, including their parameters and syntax; (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 contain less than a screenfull of information. COUSIN shows to the user information that is hidden from him by OSCON. While using OSCON the user does not see, or need to know, the structure of stored knowledge. Such information can be

∗ DOS is a trademark of International Business Machines Corporation.

discovered by the user through natural language interaction if it needs to be known. The natural language understander informs the user, in terms of English, the specific pieces of stored knowledge that are particularly relevant to some query.

(IV) Billmers & 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 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 interdisciplinary approach to consultant system design. TVX contains knowledge in two forms: abstract operating system concepts, and knowledge specific to the target system (i.e., VMS). OSCON also involves knowledge in both abstract and specific forms, distinguished by the understander and the knowledge base respectively.

LNL systems allow the user to insert queries using limited natural language. For example, an expert system called CMS-HELP has been developed by Yun & Loeb (1984) to serve 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.

The third type of consultant system are natural language understanders. The advantages of NL systems over most MENU and LNL 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).

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 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 understander and detailed knowledge in a knowledge base. Another distinction is that presently the UC program is intended to be a consultant for UNIX whereas our system is intended to be a general operating system consultant (e.g., UNIX, TOPS-20, and VM/CMS). In building OSCON 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.

Kemke (1987) describes an intelligent help system called SINIX 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

∗ SINIX is a UNIX derivative developed by SIEMENS AG.

technical aspects of the domain as well as the users view of the system. Although SC contains a knowledge base which is similar to the knowledge contained in OSCON's 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 design and approach to the UC system.

2. The problems of Transfer Semantics without inference

We have already mentioned that Transfer Semantics is a knowledge representation scheme for operating systems. However, 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. 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 embedded concept representation. So far there is no problem. The next step in the control flow of the understander is 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. This problem occurs because in each frame the postconditions are **strong** (see Mc Kevitt & Wilks 1987, p. 572). NON-DIRECTORY-FILE from the postcondition set does not match *file*† (or the embedded concept 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 an inference 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.

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 try to make the preconditions as **weak** as possible (see Mc Kevitt & Wilks 1987, p. 572). 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 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 spelling mistakes 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.

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

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

† 'More' is a command from UNIX which produces formated output on the screen.

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.

3. A language for inference rules

There are many notations 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, let’s 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 of 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 formulas 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,

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

with $H_{1,} H_{2,} \dots . \; H_n$ being general assertions means that, "given $H_{1,} H_{2,} \dots . \; H_n$ are true, then H may be deduced to be true". Also, we can define a rule of inference of the form,

$$\frac{H_{1,} H_{2,} \dots . \; H_n \,|\!- \, H_{n+1}}{H}$$

which means that, "if H_{n+1} can be deduced by assuming the truth of $H_{1,} H_{2,} \dots . \; H_n$, then H may be deduced to be true." These rules of inference are independent of the language being defined. 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 programming languages. First, lets define a language for representing actions.

3.1. A language for representing actions

We define a notation for representing operating system actions or commands. The notation,

$$\left\{ \{P\} \; A \; \{Q\} \right\} : U$$

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}. We call the information inside the bold braces (**{ }**) a *command environment*. The command environment can provide a description 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 below:

$$\left\{ \{,,,file,,/usr/paul/report,,\} \; COPY \; \{,,non\text{-}directory\text{-}file,,/usr/paul/papers,\} \right\} : User$$

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. 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 embedded concept representations. 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 paper. Let's define six inference rules to take care of the above problems and some others.

4. Some general rules of inference

In this section we look at each of the problems with Transfer Semantics and see if they can be solved by using an inference rule. Each rules is defined using the language described in the last section. There were three major problems with Transfer Semantics.

4.1. The first rule of consequence

One problem with Transfer Semantics is that postconditions specified in the postcondition set are too strong to match user queries. There needs to be some method of weakening them. Lets 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. The Transfer Semantics object hierarchy contains definitions of objects and relations between them. 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:

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

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 have,

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

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.

4.2. 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*.

In general, the rule takes the form:

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

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:

$$\left\{ \frac{\textit{plain-file} \Rightarrow \textit{file}\ ,\ \{,,\textit{file},\}\ \textit{LIST}\ \{Q\}}{\{,,\textit{plain-file},\}\ \textit{LIST}\ \{Q\}} \right\} : \textit{User}$$

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 "=>".

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, i.e. because device-file => file is true this does not mean that file => device-file is also true. However, implication is transitive, 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 ‘‘=>’’ has been described before in the field of knowledge representation. For example, Fass (1986) describes this operation 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.

4.3. A theory and representation of 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 line printer ?” 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 $[A_1 < A_2 < ...A_n]$ 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 line printer ?” 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.

4.4. A rule of composition

A third power problem with Transfer Semantics is that sometimes people like to mention more than one action in a query. In the query, ‘‘How do I detect misspellings in a file and more them ?’’ there are two actions mentioned. 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 (SPELL & PRINT) 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 as follows:

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

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 example query is:

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

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 defined at a more detailed level in the dynamic knowledge base. The dynamic knowledge base is the heart of the knowledge base for OSCON. Hegner (1987) describes an ‘interconnection calculus’ for connecting commands and one type of interconnection called ‘sequential composition’ is dealt with. Any complex object may be viewed as a serial interconnection. This serial interconnection is exactly the type of composition described above. We are pointing this out because the parallel between the *rule of composition* of the understander and the knowledge base *interconnection calculus* demonstrates one of the most important principles of our system. That principle is to place abstract knowledge about operating systems in the understander, while keeping detailed information in the knowledge base.

The rule of composition is an abstract representation defining of a mechanism which composes 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. Those with only one command are called *single-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 planner PlanCon.

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 system involving any sequence of commands. The user or the system would be able to determine if the sequence in mind was productive or detrimental. PlanCon will be very useful for

OSCON too. 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.

4.5. 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 is applied, "How do I append the file mbox to /usr/paul/post ?" In this case the user wants to append one file to another. Lets 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:

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

This general formula states that if {P} A {Q} is true and {S} A {R} is true then it is possible to infer {P /\ S} and {Q /\ R} to be true too. Here is a more specific formula for the example query above:

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

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.

4.6. 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:

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

This general formula states that if {P} A {P'} is true and {Q} A {Q'} is true then it is possible to infer {P ∨ P'} and {Q ∨ Q'} to be true too. Here is a more specific formula for the example query above:

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

From the above specific 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.

The AND and OR rules are different to 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 into a frame. 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. There is a difference between the AND and OR rules in that the AND rule is defined because it is necessary, whereas the OR rule is defined for efficiency reasons. 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. Thus the AND rule adds together necessary conditions on frames while the OR rule OR's together optional conditions on frames.

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 an interdisciplinary approach to operating system design.

4.7. The No-consequence rule

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

{P} A {P}

The rule shows us that after executing an action A the preconditions do not change at all. A command such as ‘‘who’’ in the UNIX operating system could be considered under a no-consequence rule because it does not really change the states of objects or data in the system. Any no-consequence rule can be executed a number of times throughout any command environment without having any effect on objects within that command environment. 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 commands it just applies the no-consequence rule and does not change the precondition set. 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 operate 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 members **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 OSCON 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 has not been investigated yet.

5. Relation to other work

The action frames for Transfer Semantics are described as plans in much of the literature (see Fikes and 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 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. They define a semantic structure for describing interactions between actions, both concurrent and sequential, and for composing simple actions to form complex ones. Again, this work relates to the function of PlanCon.

The UNIX Consultant (UC) program (Wilensky et al. 1984, 1986) has various elements of inference embedded within the system. 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 paper. 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 PlanCon where a number of command environments are concatenated to produce some effect. We differ from the UC approach as in PlanCon goals and plans are 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 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 i.e. (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 processes in PlanCon. 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 (1987) contains a rich knowledge base similar to our Transfer Semantics component. 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 should be able to 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) has interesting similarities with 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. Of course, 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 formalizing 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.

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 function of PlanCon.

6. Conclusion

In this paper we have described the architecture of an operating system consultant called OSCON, and more particularly, the operations on a knowledge representation. 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 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 the semantics because of its clarity and coverage.

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. This is done because if all conditions on frames are specified the frames would become too large. The inference rules allow the system to infer all those things that otherwise would not be represented in frames.

We hope to include a learning component called LeCon at a later stage. LeCon will be similar to the component called UTeacher in the UC system (Wilensky et al. 1986). LeCon would allow any user to update knowledge in OSCON 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. 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.

Our approach brings up the question of procedural versus declarative representations. The inference rules are procedural and Transfer Semantics (action frames + object frames) is declarative. However, and this is an important point, the very fact that we have defined clearly and precisely what we are doing procedurally with six inference rules means that the procedural elements in the system are now, ‘‘declarative’’. That is exactly why we have defined the rules explicitly. The reader must trust us to program the rules as we have stated them. Therefore, one major conclusion of our work, is that procedural representations can be made declarative if one takes the time to do so.

Although six general rules of inference have been defined here, the rules may need to be extended for certain types of each natural language form. Inference rules could be thought of as having their own hierarchy just like a semantic network or inheritance hierarchy, or whatever. Such hierarchies of component goals and actions for domain-dependent plans have been used by Carberry (1983). Litman & Allen (1984) and Pelavin & Allen (1987) have also developed a model based on a hierarchy of plans and metaplans.

Take for example the Rule of Composition, one of the more complex rules that does a lot of work. If a user asks something like, ‘‘How do I spell a file and print the mistakes on the Imagen ?’’ there is a composition of the commands spell and print. In this case the frame for SPELL is embedded inside that for PRINT. For the query, ‘‘How do I print a listing of my directory ?’’ the LIST frame is embedded inside the PRINT frame. These forms

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

of queries are examples of simple serial interconnection where the output of one is passed as the input to another. However, that will not be good enough for 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 KILL frame just needs 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. And, again there will surely be many ways of sifting information from commands. One can easily see that it would certainly be naive to suspect that six general rules of inference will be enough. Our next job is to locate and specify new and interesting rules of inference.

Elements of the Rule of Composition 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 also be generated by PlanCon.

In this paper 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 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 OSCON to model the interaction of system and user planning on the basis of differing beliefs and plans.

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 we have made a start in answering this question and many more.

Acknowledgements

We wish to acknowledge Yorick Wilks, Jerry Ball and Dan Fass from the Natural Language Group at the Computing Research Laboratory for providing interesting comments on this work. Dr. Stephen Hegner at the University of Vermont also contributed salient points.

References

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

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.

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

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

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

Carberry, Sandra (1983) *Tracking user goals in an information-seeking environment*. Proceedings of the National Conference on Artificial Intelligence (AAAI-83), University of Maryland, Washington, D.C.

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

Fass, D.C. (1986) *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.

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

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

Gabrielan, A. & Stickney, M.E. (1987) *Hierarchical representation of causal knowledge*. Proc. Western Conference on Expert Systems, July 2-4, 1987, Disneyland Hotel, Anaheim, California.

Goldstein, I.P., & Roberts, R.B. (1977) *Nudge, a knowledge-based scheduling program*. In Proc. IJCAI-5, 257-263.

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

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

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

Hegner, Stephen J. (1987) *Representations 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.

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

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

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

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

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

Litman, Diane J. & Allen, James F. (1984) *A plan recognition model for clarification subdialogues*. Proc. 10th International Conference on Computational Linguistics, and, 22nd Annual meeting of the Association for Computational Linguistics (COLING-84), Stanford, California, 302-311, July.

Martin, Paul; Appelt, Douglas & Pereira, Fernando (1983) *Transportability and generality in a natural-language interface system*. In Bundy, Alan (Ed.) Proc. IJCAI-8, Karlsruhe, West Germany, August, 573-581.

McDonald, James E.; Stone, J. D., & Liebelt L. S. (1983) *Evaluating a method for structuring the user-system interface*. Proceedings of the 27th Annual Meeting of the Human Factors Society, 834-837.

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

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

Mc Kevitt, Paul & Wilks, Yorick (1987) *Transfer Semantics in an Operating System Consultant: the formalization of actions involving object transfer.*. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87), Vol. 1, 569-575, Milan, Italy, August.

Mc Kevitt, Paul (1986a) *Parsing embedded queries about UNIX*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-72, Rio Grande Research Corridor, Computing Research Laboratory, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

Mc Kevitt, Paul (1986b) *Selecting and instantiating formal concept frames*. Memoranda in Computer and Cognitive Science, Memorandum MCCS-86-71, Rio Grande Research Corridor, Computing Research Laboratory, Box 30001, New Mexico State University, Las Cruces, NM 88003-0001.

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

Owicki, S. & Gries, D. (1976b) *Verifying properties of parallel programs: an axiomatic approach*. Communications of the ACM, Vol. 19, 279-285.

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

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

Pollack, Martha E. (1986) *A model of plan inference that distinguishes between the beliefs of actors and observers*. In Proc. of the Association of Computational Linguistics Conference, 207-214.

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

Sparck-Jones, Karen (1983) *Shifting meaning representations*. In Bundy, Alan (Ed.) Proc. IJCAI-8, Karlsruhe, West Germany, August, 573-581.

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

Tyler, Sherman W. & Treu, Siegfried (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, 53-60, Summer-Fall.

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

Waltz, David (1975) *Natural language access to a large database: an engineering approach*. Advance papers IJCAI-4, Tbilisi, Georgia, USSR, Sept, 868-872.

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

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

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

Wilensky, Robert (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.

Wilensky, Robert (1987) *Some complexities of goal analysis*. Preprints of Conference on Theoretical Issues in Natural Language Processing-3 (TINLAP-3) Computing Research Laboratory, New Mexico State University, 97-99, January.

Wilks, Yorick and Ballim, Afzal (1987) *Multiple Agents and the Heuristic Ascription of Belief*. In Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-87), Milan, Italy, Vol. 1, 118-124; also as CRL Memoranda in Computer and Cognitive Science, MCCS-87-75; and to appear in Noel E. Sharkey (Ed.), Advances in Cognitive Science, Chichester: Ellis Horwood.

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

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

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