Artificial Intelligence Review **14:** 121–141, 2000.
Stephen J. Hegner et al. (eds.), Intelligent help systems for UNIX.

USCSH: An Active Intelligent Assistance System

MANTON MATTHEWS*, WALTER PHARR[1], GAUTAM BISWAS[2] and HARISH NEELAKANDAN
Department of Computer Science, University of South Carolina, Columbia, SC 29208, USA, matthews@cs.sc.edu; [1]*Current address: College of Charleston, Charleston, SC 29424, USA;* [2]*Current address: Vanderbilt University, Nashville, TN 37235, USA (*author for correspondence)*

Abstract. This paper describes the knowledge sources and methods of knowledge acquisition for USCSH (University of South Carolina SHell). USCSH is an active intelligent assistance system for Unix.[1] The system operates in two modes, the active mode and the intelligent mode. In the active mode USCSH monitors the user's interactions with the system, and at appropriate times makes suggestions on how the user may better utilize the system to perform tasks. In the intelligent mode the system accepts questions in natural language and responds to them, taking into consideration the ability of the user and the context of the question.

Keywords: assistance systems, shell, user-modelling

1. Introduction

New software systems are constantly increasing in functionality, and the continued increase in hardware capability-to-cost ratio will serve to make the delivery of more sophisticated software possible. This makes an assistance system an essential component of these complex systems. Traditional approaches to assistance generally fall into one of several categories: command indexed systems (e.g., Unix man), hierarchical systems (e.g., VMS help), and tutorials (e.g., Unix learn).

Regardless of the category these traditional approaches to assistance generally suffer from several deficiences. Responses are too verbose and are not adapted to the user's level of knowledge of the system. On-line manuals must explain everything, and novices find them confusing, while more experienced users find it quite annoying to have to browse through a lot of irrelevant material to find the information they desire.

Moreover, the systems only respond when the users realize that they need to know something, and only after they figure out the right keyword to use in the query (which places the burden of translating a request for specific help into command names on the user), or by browsing through menus (which

can be quite time consuming). The problem with interactive tutorials (e.g., the Unix **learn** feature (Kernighan and Lesk 1976)) is that they are primarily designed for novice users, so they are rarely used for reference. In the day-to-day rush to get things done the user rarely finds time to devote to a "learning experience".

2. Active and Intelligent Assistance

Recently there has been substantial research addressing these deficiencies. The new approaches to assistance systems can be classified into two categories, passive and active, depending on how the interaction with the user is initiated. A passive system responds to user initiated queries, while active systems will at times initiate the interaction with the user.

The idea is to emulate human consultants, who consider user proficiencies and the current context of their activities in formulating responses to queries. In addition, consultants aid a user's learning process by indicating better and more efficient ways of getting a task done, especially when they notice the user doing things in an inefficient manner. To achieve such capabilities, on-line systems need to:

1. model individual users, i.e., keep track of their strengths and deficiencies, and use this information to occasionally make suggestions and help users improve their capabilities,
2. determine the context of users activities, and ensure responses cater to their individual needs and proficiencies, and
3. provide means so that user queries to the system need not be constrained by system commands and terminology.

In our research, we adopt a knowledge-based approach in the design of USCSH, an active assistance interface for the Unix operating system. This system addresses the capabilities mentioned above, and therefore in many ways mirrors the capabilities of intelligent tutoring systems which explicitly structure, monitor and control the student's learning processes (Clancey 1982). However, USCSH is designed to assist users and help improve their capabilities while they are involved in day-to-day activities, rather than only when they are in a tutorial environment. This requires that the system be unobtrusive, and not ask users too many questions, or provide unnecessary prompts while they are involved in routine activities. Therefore, our system differs from many tutoring systems (e.g., UMFE (Sleeman 1985)) which interrogate users to determine their proficiency on different system concepts while formulating responses to their original queries. This idea is reiterated in the design and implementation of computer coaching systems (Burton and Brown 1982).

One of the major deficiencies of existing assistance systems is that users must know "what they need to know," or at least must know command names or keywords that index into the information they need. Unix, as many have observed, does not use mnemonic names, and this makes it difficult and frustrating for new users to discover command names such as "cat" or "ls". Natural language assistance systems overcome this by allowing users to phrase queries in more general terminology. The goal of such systems is to provide more flexibility, and allow the handling of terse requests such as "help on deleting lines." There has been considerable work and progress in natural language assistance systems.

With current complex systems many beginning users learn to use a minimal set of capabilities and are content to remain at this level of performance. For example, it has been observed that experienced Unix users know a small percentage of the commands available to them (Hanson et al. 1984).[2] Their learning curve reaches a plateau and does not progress further because they feel that "they know how to use the system." Their knowledge of the use of the system does not progress until it is triggered by another "nugget" of information (Matthews and Nolan 1985; Fischer et al. 1985). This trigger often occurs when observing another user, when another user makes suggestions or comments, and by casual (accidental) reading while looking for other information.

3. Criteria of System Design

Two important criteria need to be discussed in assistance system design. The first criterion specified by Borenstein (1985), addresses the degree to which the system facilitates the accomplishment of a particular task by a user who does not currently know how to do it. The second criterion, which is equally important, refers to the effectiveness of the system in aiding users' learning processes by providing mechanisms that enable them to improve their level of performance, as they are involved in their day-to-day activities on the system. Frequently the complexity of most interactive software systems (Unix, the vi editor, etc.) makes it a difficult task even for regular users to master the complete functionality of the system. Users pick up a small subset of capabilities and commands provided by the system so as to get by in performing routine activities, and because of other work pressures remain content to use the system at that level. Only when they encounter a problem that cannot be solved using their current set of known commands do they consult other sources, such as manuals and more experienced users, to learn about capabilities and features they were previously unaware off. Users may also learn additional features and capabilities by chance meetings and dialogues

with more experienced users, or even by observing more experienced users at work.

LEVI (LEarning VI) (Matthews and Nolan 1985; Matthews and Biswas 1985; Biswas et al. 1985), the first prototype active assistance system that we built at the University of South Carolina, addressed some of these issues. LEVI is an active assistance interface for the "vi" editor on the Unix system. It maintains models of individual users in terms of their proficiency in the usage of vi commands. The user model is an overlay model with the underlying expert model being a functional decomposition of vi capabilities. The tutoring component provides assistance to users in the form of prompts that are triggered by a set of scripts that identify some user misconceptions and inefficiencies in performing tasks, and a prompting mechanism that indicates more efficient ways of achieving the same task. The system also uses periodic prompts at regular intervals to make users aware of system capabilities they did not use in the past. Using the user model as a guide, LEVI selects periodic prompts which guide the user to new plateaus in performance and gradually would expose him to the entire domain. The goal was to have "an expert system in the interface" that would monitor the user interaction and guide the user in becoming more proficient in the use of the system.

Based on these criteria and our experiences with LEVI, we feel that an effective assistance system should be developed using a framework that includes:

1. an *ideal* model that is used to explain system concepts and tasks to users,
2. a *user* model that keeps track of what the user is doing, and what he knows and does not know,
3. a *natural language interface* which can understand users' queries posed in natural language, so that users are not constrained by system terminology and rigid query formats, and
4. a *response generation* mechanism that checks the user proficiencies and the context of the user's activities before responding to a user query or prompting the user.

The system architecture of USCSH is based on these principles.

4. Overview of USCSH

The goal of the Active Assistance project at the University of South Carolina is to develop an active and intelligent user interface for the entire Unix system, which can overcome the shortcomings of current assistance systems. The aim, as with LEVI, was to have an expert system interface that would monitor the user interaction and by making appropriate suggestions guide the user in becoming more proficient in the use of the system. The second

prototype system developed in this project is 'USCSH', the University of South Carolina SHell (Matthews and Biswas 1985).

This active intelligent assistance interface, residing between the user and the Unix operating system, operates in both active and passive modes. In the active assistance mode the system continuously monitors user interactions with the Unix system, maintains user models, and at appropriate times makes useful recommendations to users about system capabilities in the context of their current activities.

User commands are directly passed through to the Unix system and also posted on the assistance system working memory. The knowledge base for the overall system consists of a system description component and a comprehensive user model. The system description component contains a hierarchical organization of Unix shell (e.g., ls, chmod, cd) and utility (e.g., vi, mail, nroff) commands, typical tasks that users perform on the system (e.g., creating a document, debugging a program), and Unix system concepts (e.g., the concept of a pipe, the hierarchical nature of directories, or the role of the buffer in the vi editor). The lowest level in the hierarchy corresponds to specific commands and task names, whereas higher level nodes represent more general concepts and task descriptions.

One of the key components of USCSH is the comprehensive user model. The central representational component of the user model is an overlay model (Carbonell 1970; Carr and Goldstein 1973; Clancey 1982), that is basically a collection of nodes that directly correspond to the nodes of the system capability hierarchy. A number of parameters, such as user proficiency and date last used, are associated with each node. In addition to the overlay structure, the user model contains a set of rules and procedures that are used for deriving user proficiencies when explicit information is not directly available from the user interactions. These rules are based on the Unix system hierarchy and assumptions made about a user's typical learning process.

A second component of the user model is an executable process model which represents the ideal and non-ideal models of user behavior. The ideal model contains plans that describe how more complex tasks can be executed as a sequence of simpler tasks and system commands and is represented as a hierarchy of frames (Minsky 1975). The non-ideal model (also called a buggy model by Brown and Burton (1978)) records plans that identify user errors and misconceptions which result in their performing tasks in an incorrect or inefficient manner. Based on sequences of user inputs and the (Unix) system's response to these inputs, this model is used to infer what tasks the user is involved in, and whether he is making errors or choosing inefficient methods in executing these tasks.

The active assistant is involved in two parallel but related activities as it tracks the user's interactions with the system. First, it continually updates the dynamic user model to reflect the extent of his knowledge of the system, Second, it prompts the user *on its own initiative*, either periodically, or if it determines that the user is performing a task incorrectly or inefficiently.

In the intelligent assistance mode the system responds to user queries on how to use the system. The first versions of the intelligent assistant were not full-fledged natural language processing systems, but used the menu/hypertext-like Screen Display System. The later version, written in LISP, uses an ATN parser to understand natural language queries.

The intelligent assistance mode of USCSH is activated when the user types in the command "query", which is treated as another C-shell command. The ATN-based parser integrates syntactic and semantic information, processes the query and translates it into an internal frame-based representation. The response generation mechanism interprets the query in the context of the user's current activities and then generates responses that take into account the user's proficiency in the use of the system. Response generation primarily entails matching the internal representation of the query against the frames that represent the plans of the ideal model, and retrieving a suitable frame or set of frames that contain the response to that query.

5. Active Mode

There are two implementations of the active components of USCSH. One version, implemented in OPS5, runs as a separate concurrent process which communicates with the csh through a pipe. This first version illustrated the concepts but was slow (it took up to 3 minutes overhead on an NCR Tower), so we looked for ways to improve the performance. The second version is implemented by adding a "goal parser" to csh using a finite state machine implemented in C.

In the active assistance system mode user commands are directly passed through to the Unix system for execution, and to the goal parser for analyzing user activities. The goal parser recognizes both efficient and inefficient plans for achieving goals. It uses the sequence of commands observed to that point in time to update its workspace of partial matches to plans. As the user types in additional commands, some partial matches are extended, others are discarded, and some are left alone. USCSH also updates the user proficiency model, incrementing nodes corresponding to observed commands, goals and concepts. After recognizing an inefficient sequence, USCSH determines if at this time it should become active and prompt the user. If the user model indicates that the user knows about what the system is about to suggest then

the system will refrain from making that suggestion. This is illustrated in the third example in Section 5.1.

5.1. *Examples of user interactions with USCSH (active component)*

Example

```
RECOGNIZING USER INTENTIONS

% vi prog.c
% cc -o prog prog.c
% prog
% model c_programming
```

After this sequence of commands USCSH recognizes the goal “c-program development.”

Example

```
RECOGNIZING WITH NOISE

% vi prog.c
% cd Maildir
% mail tech
% cd
% ls
% cc -o prog prog.c
% prog
```

Even with interspersed “noise commands” USCSH recognizes the goal “c-program development.”

Example

```
INEFFICIENCY MONITORING: FILENAME EXPANSION

% rm test.o
% rm test1.o
% rm test2.o

If the directory containing the files contained no more
``.o'' files then any appropriate prompt might be
You can use the metacharacter ``*'' in the command rm *.o
instead of the commands
```

```
rm test.o
rm test1.o
rm test2.o
```

Note that if there were more ".o" files in the directory or if the user was very familiar with the use of "*" then this suggestion would not be appropriate and would not be given. If there were more ".o" files in the directory then a different pattern might be used (if there is a sufficiently simple one).

In addition to inefficiency prompts USCSH will at user-configurable intervals give periodic prompts. Periodic prompts are designed to gradually lead the user to new plateaus of performance in using the system. The idea of the sequence of plateaus of performance that the user is guided through is not unlike the "increasingly complex micro-worlds" of Fischer (1985). However, in USCSH we use both the inefficiency, error and periodic prompts to achieve this guidance where Fischer's systems rely only on error/inefficiency messages. Our inclusion of the periodic prompt is primarily in the belief that inefficiency models can never be complete. Also, note that there may be some tasks that the user achieves via another mechanism altogether because he is unaware of capabilities of the system. An example would be using another system such as a MacIntosh to prepare transparencies, because one does not know of the capabilities of TEX and troff.

6. The User Model for USCSH

The primary purpose of a user model is to represent what a user knows and what the user does not know. The user model of USCSH includes:

1. an overlay model which describes the system capabilities and concepts.
2. a process model which recognizes inefficiencies and higher level goals.

Rich (1983) has presented a number of characteristics (or dimensions) that can be used to classify user models:

1. models for a single *stereo-type user* versus a *collection* of models for *individual users*,
2. models that are specified *explicitly* by *users or system designers* versus models *inferred by the system* based on the user's interactions with the system, and
3. models of *long-term* user characteristics versus models of the *current* or *more recent* user activities.

Sleeman (1985) added a fourth characteristic that is based on the nature and form of the information contained in the user model, and the reasoning processes used to infer that information. For USCSH to achieve the desired level of sophistication, it needs to have models for individual users and not

just stereotypes. Furthermore, since it is designed to be an assistance system that aids users *while they are engaged in their regular activities*, the model should be derived almost entirely by observing individual user interactions with the system.

7. The Overlay Model

The overlay model that is used in USCSH is an overlay on a functional decomposition of capabilities and concepts. It is basically a hierarchy of nodes, with each node representing what the user knows about the system capability or concept that corresponds to this node. The hierarchy is arranged with more abstract concepts/capabilities higher in the hierarchy and more specific concepts/capabilities further down in the hierarchy. Each Unix concept/capability has a corresponding node in the hierarchy that indicates how much the user knows or does not know about this concept/capability. Also, associated with each node in the hierarchy are two numbers that reflect the difficulty and importance of the concept/capability. The hierarchy and the importance and difficulty numbers are independent of the user and are fixed during the construction of the system.

Each node of the overlay model contains a slot that stores an individual user's proficiency in the use of the concept, command or task represented by the node. Also associated with each node are two numbers that reflect the difficulty and importance of the concept associated with the node. These numbers and the links in the hierarchy are static and independent of the user, being determined during the construction of the system. The individualized portion of the user model is the proficiency or mastery rating associated with the nodes.

The proficiency ratings of the nodes for commands are updated by directly observing the use of these commands, giving an observed rating, r_o. When the user is observed to be using a simple command correctly or incorrectly, the corresponding proficiency is incremented or decremented, respectively. The proficiency ratings of higher-level nodes that correspond to abstract concepts in the domain are *derived* using the system hierarchy and the importance and difficulty factors associated with the concepts, yielding a derived rating, r_d. For a given node, r_d is computed as a weighted sum of the ratings of its children. However, there exist some nodes in the hierarchy which could have both an observed and a derived rating associated with them. For example, the node associated with the concept of deleting a line in the visual editor "vi" has an observed rating which is based on the actual use of the "dd" command by the user. At the same time, it is reasonable to assume that a user who is very proficient with the visual editor also knows the concept of deleting a

line, and though the system may not have observed the user performing "dd" commands, a derived rating, $r_d(dd)$ would be associated with the "dd" node. This number $r_d(dd)$ is a function of $r_o(vi)$.

Ratings are integers in the range –128 to 127; 127 represents complete belief that the user knows the particular concept, –128 represents complete belief that the user does not know the concept, and 0 implies that the system has no information on the user's proficiency in this concept or task. Also associated with each node are threshold intervals which depend on the importance and difficulty of the concept. The extent of the user's expertise with the concept is indicated by the interval to which the rating belongs, this information being used by the assistance system to tailor its prompts and responses to the user. The first threshold interval represents a "no-information" interval, i.e., nodes whose ratings belong to this interval are nodes about which the assistance system believes it does not have sufficient information. For nodes with both r_o and r_d, first r_o is used, and if this provides no information, then its derived rating r_d is computed and used.

The system, on its own initiative, periodically makes suggestions on topics that it feels the user needs to know. This is done by searching the overlay model for the "best" topic to prompt on. The best prompts, of a number of suggestions that may appear equally appropriate, are those that fill in the largest gaps in the user's conceptual model of the system (Fischer et al. 1985), i.e., those that suggest new features and functions that the user is currently unaware of, and at the same time are related to his current tasks and activities. There are two decisions to be made in generating a periodic prompt: *when* to prompt, and *on what* to prompt. The *when* decision for periodic prompts is controlled via user specified parameters. The range of values this parameter can take varies from very frequent (one every half hour of connect time) to very infrequent (one every two weeks). Also, the method of delivery of the prompts can be chosen by the user. Periodic prompts may be displayed on the screen either at the start or end of a session, delivered via electronic mail, or stored in a suggestions file that users may browse at their leisure. The decision of *what to prompt on* is implemented using a recursive algorithm that starts at the top node of the system hierarchy and proceeds down into lower levels in a depth first manner, at each step selecting the "most promising" branch to pursue. The evaluation of each node for this purpose is based on its proficiency ratings, its importance and difficulty numbers, and the relevance of the associated concept to the user's context of activities, which is determined by the ideal model. The first node encountered that has observed proficiency below a prespecified threshold is selected.

Periodic prompting has been incorporated in USCSH, because prompting based simply on detection of incorrect or inefficient plans (which is the other

occasion when the system prompts the user, and is discussed in the next section) is necessarily incomplete, and cannot cover all aspects of the assistance domain. Periodic prompts inform the user about system capabilities that he is not aware of, thus providing him a vehicle for expanding his conceptual model of the system and improving his overall performance. At the same time, periodic prompting, unless carefully controlled, could be an annoyance to any user, and it is for this reason that we leave the final decision about the prompting frequency to the user.

8. The Process Model

The process model of USCSH has two components. The ideal component contains a set of rules that specify higher level goals such as "development of a C program" and "creation of a document". The non-ideal component contains a set of rules to specify inefficient command sequences and erroneous commands. It also recognizes errors that are correct commands of other systems, such as 'dir' for VMS, DOS, CP/M.

The ideal and non-ideal components of the process model have been developed by a detailed analysis of the day-to-day tasks that users perform on the Unix system. This can vary a great deal depending on the type of user and the kind of environment he works in. USCSH is currently tailored for users in typical office environments, who use the system mainly for document and memo preparation, communicating with other users, and probably maintaining a calendar of activities. This determines the type of Unix utilities that need to be included in the ideal model: the basic shell, file system, the visual and line-oriented editors, text formatters, the mail system and the calendar utility.

The ideal component of the process model contains plans that describe how more complex tasks can be accomplished using a sequence of simpler tasks and commands, and has been implemented as a hierarchy of frames. For example, consider the task of "moving a block of lines in a file" using the visual editor. There is a frame associated with the concept of **move**. This frame contains a number of slots that represent the various objects in the domain which can be subjected to a **move**, e.g., files, and lines in a file. Each slot has a facet called **command** or **task**. A **command** slot means that the required action can be performed using a simple command, whereas a **task** slot implies that the particular task is executed as a sequence of sub-tasks. (A sub-task could be either a simple command or another sequence of sub-tasks.) The value associated with this facet represents either the simple command or the sequence of sub-tasks that have to be performed, as the case may be. In our example, the **line** slot has a **task** facet which contains the three elements

(delete lines), (position cursor) and (put lines). Plainly stated, this plan says that "In order to move a block of lines in a file, delete that block, place the cursor to the desired new position, and put the block at this new position".

Now in order to determine completely how to move lines, the inferencing mechanism has to recursively examine the frames for the three new tasks, i.e., **delete**, **position**, and **put**, which will in turn contain the information regarding how those actions can be performed. This process thus corresponds to the traversal of a tree rooted at the concept that we started off with, viz., **move**, in this case, until we reach all the leaf nodes of the tree, i.e., frame-slots with only **command** facets.

The information represented in the frames can be also used in reverse to determine what activity the user is currently involved in. For example, the frames discussed above tell us that if the user is observed to delete a block of lines, re-position the cursor, and then put that block while using the visual editor, then we can conclude that the user is moving a block of lines in a file. As another example, if a user goes through an *edit-compile-run* sequence of commands, then it can be inferred that he is trying to debug a program. Thus the ideal component of the process model can be used to establish the context of the user's activities. This is important because, first, the prompting mechanism has to select a prompting topic that is *most relevant to the context of the user's current activities*, and second, the intelligent assistant needs to refer to the context to resolve ambiguous references during query processing.

The non-ideal component of the process model represents typical user errors and inefficiencies in executing tasks, and can be looked upon as a list or catalog of "bad" plans. Some of the user inefficiencies and errors have been derived by empirical observation; others were derived by pooling together our own intuitions and experiences in using the system.

9. The OPS-5 Version

The first version of the process model of USCSH was implemented in OPS-5, where the ideal and non-ideal plans are represented as OPS-5 production rules. The antecedent of a typical rule is the occurrence of a particular sequence of commands, and the corresponding conclusion is the recognition of either the context of the user's activities, or an incorrect or inefficient usage of commands. An example of an inefficient plan to rename a file is given below:

Example

```
cp <file1> <file2>
rm <file1>
```

```
->
inefficient; You could have used the mv command.
```

The above rule states that if the user makes a copy of a file, and then removes the original, then it would have been easier to use the rename command on the original file. However, this rule does not always apply, since there are occasions when the user has no option but to make a copy and then remove the original (e.g., when the user wants to change the ownership of a uucp file, or when the file indicated by the path'file1' has multiple links). How this possibility is handled is explained later in this section. Another inefficient plan, which pertains to the visual editor states that "if the user uses the **dw** command (to delete a word) immediately followed by the **i** command (to insert text), he could have been more efficient and used the **cw** command (to change a word)". It is clear that for the above rules to fire, the commands should satisfy certain sequencing requirements depending on the rule. This sequencing information is extracted from the user's interaction through the use of "timetags".

The OPS5 process is created as soon as the user logs on, and runs concurrently with the Unix shell. Information exchange between the shell and the OPS-5 process is accomplished via two-way pipes. Every command that the user types in is passed to the OPS-5 process as a working memory element in an "argc-argv" format that includes the command name, number of arguments, and the arguments themselves. Also associated with each command is a time-tag which indicates the recency of the command in the current session. This representation scheme thus gives us the *exact sequence of commands* that the user has used in the current session. The sequencing requirements that commands have to satisfy before they can qualify to participate in an inefficient plan are checked by comparing the time-tags of the commands. The comparison is done by *LISP functions* called from within OPS-5. Thus, the rules contain calls to external LISP functions which take the time-tags of the commands as arguments, and check if they satisfy the appropriate sequencing requirements. Other LISP functions are used to compute the length of an argument, look for arguments which have a common suffix, and identify path-names which have identical tails (e.g., /usr/accounts/grad/phil/report and report).

The firing of a buggy rule implies that the user is performing a task inefficiently, and the prompting mode is activated. The user model is examined to determine whether the user is proficient and also whether he has already been prompted on this topic. In either case, the prompt is not displayed. For instance, if the user is known to be proficient at the "mv" command, and he still makes a copy and then removes the original, then he is perhaps trying to change the ownership of a uucp file, and it would be inappropriate to prompt

him about “mv”. Context analysis may also help to corroborate this assumption. If a prompt is generated, then it is tailored to the level of proficiency of the user which is determined by the threshold interval in which the user’s proficiency lies. Thus we ensure that active prompting is appropriate to the user’s expertise and context of activities, maximizing the benefit derived from active prompting.

The attachment of time-tags to commands provides flexibility in matching sequences of commands to goals, or users’ tasks. This is because “noise” commands that occur between commands in the actual sequence should be ignored. For example, in between the edit, compile and run sequence, the user may use a ‘ls’ command to list the contents of his directory, or use the ‘mail’ facility to communicate with other users. However, these intermediate commands are actually “noise” commands, since they do not actually alter the user’s goals, though they appear in the actual sequence. On the other hand, there are instances where intermediate commands are significant. For example, during the edit, compile and run sequence discussed above, if the user does a ‘copy’ (cp), or a ‘move’ (mv) into the current file, the conclusions of the plan sequence given above may no longer be valid. Such instances are specifically coded into the plan to inhibit its activation.

10. The Goal Parser Version

Implementation of the process model as a OPS-5 production system enabled the development of a quick prototype for purposes of testing and analysis. However, as the number of plans in the ideal and deviant models kept increasing, there was a noticeable degradation in performance. For example, with approximately 50 ideal and deviant plans, it took about four to five times more time for the shell to return the prompt sign for the next command, after executing a simple “ls” or “cd” command on an NCR Tower-32 system running Unix System V with three users logged on simultaneously. This prompted the design of a more efficient match scheme for plan recognition: the use of a non-deterministic finite state automaton implemented in the C language, and directly integrated with the shell. This eliminated the communication overhead that had occurred between the shell and the OPS-5 process, which was implemented as a two-way pipe.

This goal parser version of the USCSH was heavily influenced by its predecessor. The knowledge representation language is a simplified version of OPS-5, (S-OPS). The rules of S-OPS, like OPS-5, have a sequence of left-hand side elements, followed by an arrow “->”, followed by a list of actions on the right hand side. The left hand side elements are called “command instances”. Command instances consist of a command followed by a list

of predicates and functions applied to the arguments of the command. The command the user types is used to determine the state transitions in the goal parser by matching it against the next command instance in partially matched plans (rules). The system will make multiple transitions if the command matches the next command instance of more than one of the partially matched plans. The actions that can appear on the right hand side of a S-OPS rule are quite limited. The only actions possible are "goal" and "inefficiency". When either of these actions is performed the overlay model is updated and there are possible additional state transitions. In addition when a "inefficiency" action is performed, the user model is accessed to ascertain whether the user already knows the information and thus whether the prompt should really be made.

To achieve the desired efficiency we needed to avoid the obvious overhead of the pipe that connected the OPS-5 and csh in the earlier version. Thus C was the clear choice of language because of the need to integrate the goal parser with the csh. It is fairly easy to update the inefficiency/goal knowledge base. The additions to the csh that vary as the knowledge base is modified are isolated in a few modules. An additional component to the makefile for the csh, makes these modules.

A special tool was developed for generating the state transitions of the finite state automaton. This tool starts off with S-OPS productions as a representation of plans and converts them to the states and transitions of the finite state automaton. Normally each condition element of a S-OPS rule (i.e., a single component of a plan) corresponds to a state in the automaton. Since a number of different plans could have element subsequences that are identical, during the match process for identifying user goals or tasks, the automaton could be in many different states at the same time, and is non-deterministic in nature. For more details on the implementation of the automaton, the reader is referred to (Wang 1986).

In summary, the active assistant of USCSH is like a human consultant sitting beside the user, monitoring his interactions with the system. It maintains a detailed model of the user which it updates continuously. It also keeps track of the current context of the user's activities. Periodically, it informs the user about aspects of the system that he is unaware of. In deciding what topic to prompt the user on, it chooses one which might be most relevant to his current activities so as to maximize the benefits that accrue to the user as a result of learning about a hitherto unknown capability of the system. If it observes that the user is performing tasks incorrectly or inefficiently, it draws his attention to this fact, and presents a better method for achieving the same goal, but only after ensuring that the user does not already know about it. All these features make USCSH a fairly robust assistant, except for one capability, viz., being

able to respond to specific queries of users which may be posed in natural language.

11. Intelligent Mode

One of the problems with traditional assistance systems is that users often have to formulate their requests for help in terms of system commands and keywords which they may not remember or even know in the first place. This problem is made more acute in the Unix environment in which command names are not mnemonic of the actions they represent, and users find it even more difficult to remember odd-sounding command names. An effective assistant must, therefore, have the capability to interpret user queries which are posed in natural language, so that users are not burdened with the task of translating their requests in terms of system terminology.

The intelligent mode of USCSH is a natural language assistance system. It was written in Franz lisp by Harish Neelakandan. The module is compiled and then is accessed as a built-in function of csh. The parsing mechanism uses an ATN framework (Charniak and McDermott 1985) with the grammar of Winograd (Winograd 1983) as a base. The queries are processed into an internal frame structure, that is then matched against a collection of method frames to produce the response.

The natural language interface at this time is restricted to answering two large classes of queries; "how do I `<action> <objects>`" and "what is a `<concept>/<capability>.`"

Note these are the deep representations that are currently handled. There are, of course, many surface variations that get translated down to these two classes of queries. For a more detailed discussion of the Intelligent Assistant component of USCSH see (Neelakandan 1987).

The ATN-based parser converts user queries into an internal format using a backtracking parsing algorithm. The output of the parse is a *register structure tree*, which is different from the traditional phrase structure trees, and more closely related to the role structures used by Kaplan (1975). The register structure trees are represented as a collection of frames, thus, the intelligent assistant and the active assistant use a common representation scheme.

One of the principal advantages of using the ATN framework is that it allows the embedding of both general and domain-specific semantic information into the syntactic parse phase without sacrificing the generality of the parser. Moreover, it can also be used to parse incomplete and, therefore, ungrammatical input, although it has traditionally been used as a syntax-directed machine. This aspect is especially important in the domain of assistance systems, because users of assistance systems are primarily inter-

ested in getting their work done, and find it frustrating to be constrained by rigid, grammatical query formats. Thus, the ATN provides a unified representation for both syntactic and semantic processing, and this is significant both from a theoretical standpoint, as well as the from point of view of real-time natural language systems. For details of how the ATN framework has been exploited to incorporate the features discussed above see (Neelakandan et al. 1987a).

The parse query is first placed into one of various predetermined categories; these were compiled in (Pharr and Neelakandan 1986). The next step involves inferring the user's goal, since goal analysis is important in the context of user assistance systems in order to understand or interpret indirect speech acts (Wilensky et al. 1986). The user's goal is stored in a *goal-frame*. To resolve ambiguities or to make the goal more explicit, it may be necessary to analyze the context of the user's current activities. The context determining mechanisms of the active assistant are then invoked through if-needed procedures associated with the goal-frame. For example, if the user asks: *How do I move text?* then it is possible to interpret this question in two different ways. By "text," the user could either refer to a line or a block of lines within a file, or to an entire file itself. Each of these requires a different response, and therefore, the system differentiates them by storing them as distinct goals. In such a scenario, if the ideal component of the process model has determined that the user is currently in the edit mode, then it is likely that the user is referring to a moving lines within a file or between files, as opposed to the case of renaming a file or moving the file from one directory to another.

The user model may also be able to provide some clues to resolve ambiguities. For example, if the model has evidence to show that the user is proficient in moving files, then his current question is more likely to pertain to the visual editor. Thus, both the context as well as the user model play an important role during the response generation phase of the intelligent assistant. Once it has been determined that the user is referring to moving a block of lines, the next question is whether he wants to simply relocate the position of the block within the same file or whether he wants to move this block to another file. Again, the user model can be used to make this decision. It is possible that the user model does not provide much help; in that case, the response will have to state both possibilities or prompt the user for a more specific question. The possibilities are numerous, and it is impossible to handle every particular case, but if the system is aware of different possibilities, and responds accordingly even in very general terms, users would tend to have more faith in the system. A specific follow-up question could then provide more details about what the user was actually trying to determine.

After the user's goal is established, the next step is to compute a method for achieving that goal, and this is done by matching the goal-frame to the plans of the ideal component of the process model. Considering again the question about moving lines within a file, the intelligent assistant matches the goal-frame created against the frames for **move**, **delete**, **position**, and **put**, and creates a *method-frame* which contains the necessary steps required to move a line. Finally, a generation component examines the method-frame and generates an English response using generic response templates.

Throughout the generation process, the intelligent assistant constantly interacts with the user model and the context mechanism of the active assistant, and the Unix system itself in order to generate suitable responses to users' queries. We do not elaborate on these aspects here (Neelakandan et al. 1987b); presents detailed examples of how the integrated system responds to users in a more intelligent manner and provides a friendlier environment for users to work in.

In summary, the intelligent assistant plays the role of a human consultant who can respond to user queries on various aspects of the system. In doing so, the consultant keeps in mind various factors pertaining to the user and recent interactions in order to come up with pertinent responses to the queries. Therefore, users can pose fairly complex queries to the intelligent assistant *as and when they need to get more information* on specific aspects, and this helps them improve their overall performance in the use of the system.

12. Conclusions

In this paper, we have described the architecture and implementation of USCSH, an active intelligent assistance system in the Unix domain. The system uses a multi-faceted user model: an ideal model that describes the preferred way of executing tasks, a non-ideal or buggy model that describes common user errors, misconceptions and inefficiencies, and a detailed overlay model that keeps track of what a user knows and what he does not know. In addition, the system keeps track of the context of the user's activities as he is involved in his own work. Our approach is different from UC's (Chin 1986) double stereotype scheme. We feel that our system provides a much more accurate representation of individual user proficiencies on individual components of the Unix system without an excessive increase in computational costs. Besides, our model also allows us to deal with idiosyncratic users; for example, a person who may an expert in the use of the visual editor "vi", but a complete novice in using the "mail" utility.

Another feature of USCSH is that it is far more integrated with the original shell (C shell) than similar systems that have been developed. To prevent

degradation in performance with the increased computations that are required for the active assistant, we have developed an efficient matching scheme for goal and task recognition using finite state automaton techniques.

The function that we currently use for computing derived proficiencies in the overlay model are *ad hoc*; they do satisfy some intuitive requirements, but they have not been tested extensively in real environments. Feedback obtained from users of this system will provide directions for modifying and fine tuning these functions. We are now closely monitoring the performance of the current prototype to determine it's shortcomings and failures; these experiences will provide us with valuable feedback in expanding the capabilities of the process models.

We also want to move away from a completely rule-based representation of the non-ideal component of the process model. What would be more desirable is to incorporate these rules as procedural slots in the frames that we use to represent the ideal component, thus integrating both the components of the process model into one uniform representation scheme. However, this implies that we would have to do some additional book-keeping in the processing of these procedural attachments.

On the intelligent assistant side, the current implementation does not convey its own limitations to the user, and this is something which requires more work. Clearly, no natural language system is capable of parsing *all natural language*, but it would be helpful to the user if the system can give the user an idea of its vocabulary and scope of parsing capabilities especially when it fails to parse something, so that the user is aware of this and can adapt his querying style to maximize the percentage of successfully handled queries. Furthermore, we would like to eventually move away from the response templates and closer to natural language generation.

In conclusion we feel that USCSH provides a conducive environment for users to gradually increase their proficiency in Unix, as they are involved in their day-to-day activities, and the positive help provided by the assistance system encourages them to use it as often as they desire.

13. Recent Developments

Since the Berkeley workshop the team has spread out and research efforts have diversified also. The individuals of the project continue with work on modelling and intelligent agents and their applications to intelligent user interfaces. At South Carolina, Matthews has continued to lead a group working in intelligent user interfaces. With Pharr he has been worked in knowledge representation for natural language processing (NLP) and the application to dialogue systems (Pharr 1990). Matthews, Reid and Martin have worked

on a hypercube based parallel parser for natural language interfaces (Matthews and Reid 1992). Matthews and Sobczak have worked with distributed intelligent agents for spectroscopy problems (Sobczak and Matthews 1990; Sobczak 1991). Biswas heads a group at Vanderbilt pursuing a number of research directions including: modelling of physical systems, (Mosterman and Biswas 1997), intelligent manufacturing and Intelligent Learning Environments.

Notes

[1] Unix is a trademark of X/Open, Inc.
[2] To be precise "know" should be replaced by "use".

References

Biswas, G., Matthews, C. T., Matthews, M. M., Rabon, P. & Wilhite, R. L. (1985). An Active Assistance System for vi. *IEEE Intl. Conf. on Systems, Man, and Cybernetics*, 746–750.

Borenstein, N. S. (1985). The Design and Evaluation of On-line Help Systems. *Tech. Report CMU-CS-85-151*.

Brown, J. S. & Burton, R. R. (1978). Diagnostic Models for Procedural Bugs in Basic Mathematical Skills. *Cognitive Sciences* **2**: 155–192.

Burton, R. R. & Brown, J. S. (1982). An Investigation of Computer Coaching for Informal Learning Activities. *Intelligent Tutoring Systems*, 79–98.

Carbonell, J. R. (1970). AI in CAI: An Artificial Intelligence Approach to Computer Aided Instruction. *IEEE Trans. on Man-Machine Systems* **11**: 190–202.

Carr, B. & Goldstein, I. (1973). Overlays: a Theory of Modelling for Computer Aided Instruction. *Int. Jour. Man-Machine Studies* **5**: 215–236.

Charniak, E. & McDermott, D. (1985). *Introduction to Artificial Intelligence*. MA: Addison-Wesley.

Chin, D. N. (1986). A Case Study of Knowledge Representation in UC. *Proc. Human Factors in Computing Systems* **86**.

Clancey, W. J. (1982). ICAI Systems Design. *Handbook of Artificial Intelligence* **2**: 229–235.

Clancey, W. J. & Letsinger, R. (1981). Neomycin: Reconfiguring a Rule-Based Expert System for application to teaching. *Proc. IJCAI*, 829–836.

Fischer, G., Lemke, A. & Schwab, T. (1985). Knowledge-Based Help Systems. *Proc. Human Factors in Computing Systems* **85**: 161–168.

Hanson, S. J., Kraut, R. & Farber, J. (1984). Interface Design and Multivariate Analysis of UNIX Command Use. *ACM Trans. on Office Information Systems* **2**: 42–57.

Kaplan, R. M. (1975). On Process Models for Sentence Analysis. *Explorations in Cognition*, 117–135.

Kernighan, B. W. & Lesk, M. E. (1976). The Learn CAI systems. *Unix System Documentation*.

Matthews, M. M. & Biswas, G. (1985). Oracle: A Knowledgeable User Interface. *Proc. 9th COMPSAC Computer Software and Applications Conf.*, 358–363.

Matthews, M. M. & Nolan, E. (1985). LEVI: A Learning Editor. *Proc. 1985 USENIX Assoc.*, 325–331.

Matthews, M. M. & Reid, R. (1992). Parallel Parsing of Ambiguous Languages on Hypercube Architectures. *Lecture Notes in Computer Science* **604**: 539–545.

Matthews, M. M. & Biswas, G. (1985). Raising User Proficiency Through Active Assistance: An Intelligent Editor. *Proc. of Second Conference on Artificial Intelligence Applications*, 358–363.

Minsky, M. (1975). A Framework for Representing Knowledge. *The Psychology of Computer Vision*, 211–277.

Mosterman, P. J. & Biswas, G. (1997). Formal Specifications for Hybrid Dynamical Systems. *Proceedings of IJCAI-97*, 211–277.

Neelakandan, H. (1987). A Natural Language Interface to USCSH. MS Thesis, Department of Computer Science, Univ. of South Carolina.

Neelakandan, H., Biswas, G. & Matthews, M. M. (1987a). USCSH: An Active Assistance Interface for Unix. with G. Biswas, and Harish Neelakandan, *Proc. of the Intelligent Tutoring Systems 1988 (ITS-88)*, 334–341. Montreal, Canada.

Neelakandan, H., Biswas, G. & Matthews, M. M. (1987b). An Intelligent Assistance System in the Unix Domain. *Proc. Third Annual Expert Systems in Government Conference*, 55–64.

Pharr, W. (1990). A Knowledge Representation for Natural Language Understanding. *The Third International Conference on Industrial and Engineering Applications of AI & Expert Systems*, 859–865.

Pharr, W. & Neelakandan, H. (1986). Natural Language Interface for Oracle. TR, Department of Computer Science, Univ. of South Carolina.

Rich, Elaine A. (1983). Users are Individuals: Individualizing User Models. *Int. Jour. Man-Machine Studies* **18**: 199–214.

Sleeman, D. H. (1985). UMFE: A User Modelling Front-End Subsystem. *Int. J. of Man-Machine Studies* **23**.

Sobczak, R. (1991). Using a Distributed Blackboard Network for Solving Organic Spectroscopy Problems. *Proceedings of the AAAI workshop on Blackboard Systems*, July 1991.

Sobczak, R. & Matthews, M. (1990). A Massively Parallel Expert Sytem Architecture for Chemical Structure Analysis. *The Fifth Distributed Memory Computing Conference*, 11–17.

Wang, Shin-Yung (1986). A Knowledge Based Model of an Active Assistance System for UNIX. MS Thesis, Department of Computer Science, Univ. of South Carolina.

Wilensky, R., Mayfield, J., Albert, A., Chin, D., Cox, C., Luria, M., Martin, J. & Wu, D. (1986). UC – A Progress Report. BSD, Univ. of California at Berkeley.

Winograd, T. (1983). *Language as a Cognitive Process*. MA: Addison-Wesley.