Modeling UNIX[1] Command Production: What Experts Must Know

by

Stephanie M. Doane[2], Walter Kintsch,
and Peter G. Polson

Institute of Cognitive Science
University of Colorado
Campus Box 345
Boulder, CO 80309-0345

ICS Technical Report #90-1

---

[1]UNIX is a registered trademark of AT&T.

[2]This research is supported in part by contract IRI-8722792 by the National Science Foundation. We would like to thanks Anders Ericsson, Alice Healy, Andreas Lemke, and Suzanne Mannes for their valuable comments on the manuscript. We would also like to thank Gerhard Fischer and Clayton Lewis for their contributions to this work.

Correspondence should be sent to Stephanie Doane, Institute of Cognitive Science, University of Colorado, Campus Box 345, Boulder, CO 80309-0345.

Abstract

We propose a model of UNIX command production in an attempt to characterize the knowledge users of UNIX have at various stages of learning. We modeled novice, intermediate, and expert UNIX command production data collected by Doane, Pellegrino, and Klatzky (in press) with a model called UNICOM. We use the construction-integration theory of comprehension proposed by Kintsch (1988) as a framework for our analyses. We focus on how instructions activate the knowledge relevant to the performance of the task specified. We have successfully modeled major aspects of correct user performance by incorporating in the model knowledge about individual commands and knowledge that allows the correct combination of elementary commands into complex, novel commands. We show that salient aspects of novice and intermediate performance can be described by removing critical elements of knowledge from the expert model. The results are consistent with the view that the acquisition of expertise can be characterized as a increase in the number of facts known about a domain, rather than as a qualitative shift in processing strategies or knowledge structures.

Introduction

We propose to model UNIX command production in an attempt to characterize the knowledge possessed by users of UNIX. Specifically, we wish to know how people understand UNIX command production at various stages of learning. We use the construction-integration theory of comprehension proposed by Kintsch (1988) as a framework for our analyses. The construction-integration theory is a general theory of discourse comprehension that models the processes involved in reading and comprehending a text, including the activation and use of knowledge in comprehension. This model has been applied to several tasks involving discourse comprehension processes, such as story understanding, algebra word problem solving (Kintsch, 1988), sentence recognition (Kintsch, Welsch, Schmalhofer & Zimny, in press), and routine computing tasks (Mannes & Kintsch, 1989).

In the present application, the texts to be understood are brief instructions to produce legal UNIX commands. The focus of our analysis is not so much on understanding the text per se, but on the way these instructions activate the knowledge relevant to the performance of the specified task. Understanding in this context implies the ability to plan and execute a series of actions. The instructional text serves as the cue for retrieval of the relevant knowledge, and for organizing this knowledge to produce an action sequence.

Our analysis of the knowledge structures maintained by users of UNIX with varying levels of experience and the way this knowledge is used in composing UNIX commands is meant to clarify some controversial questions concerning the development of expertise. We are concerned with whether the development of expertise is best characterized by a qualitative shift in knowledge organization and processing, or whether it can be characterized by the monotonic increase in facts accumulated with expertise, as argued by Wertheimer (1945) and Simon (1986), respectively.

Evidence for both viewpoints can be found in the literature. Numerous researchers have suggested qualitative differences between experts' and novices' mental representations and information processing. Much of the evidence for these qualitative differences comes from research in the domains of physics (e.g., Larkin, 1983; Larkin, McDermott, Simon, & Simon, 1980; Chi, Feltovich & Glaser, 1981) and computer programming (e.g., Adelson, 1981; 1984; McKeithen, Reitman, Rueter, & Hirtle, 1981; Jeffries, 1982). This research suggests that experts form more abstract mental representations which are qualitatively different from those of their novice counterparts, as indicated by their verbal reports about how they approach physics problems and computer programs. Furthermore, these researchers conclude that the expert tends to have a "deep" representation, whereas the novice has a "surface" representation (Chi, et al., 1981). They find evidence in the semantically based performance of expert programmers and principle-bound performance of expert physicists, as contrasted with syntactically based performance of novice programmers and physically-bound performance of novice physicists. Put in other terms, they suggest that the expert represents knowledge in functional units, while the novice representation is made up of individual, discrete pieces (Chi, et al., 1981).

The observations made by these researchers are not in question. What is debatable is whether these observations are evidence for a qualitative shift in thinking, or whether they can be accounted for by a monotonic increase in knowledge about a domain. Some researchers have suggested that the development of expertise can be explained by the acquisition of schematic knowledge structures (e.g., Koedinger & Anderson, 1989;

Schank, 1982). These additional pieces of knowledge, then, lead to expert performance -- not a qualitative shift in organization. In complex systems merely adding some new pieces may have dramatic effects on the actual behavior of the system, making it look like a profound reorganization of the system has occurred. Our work addresses this issue concerning the development of expertise. It suggests that some aspects of the development of user expertise with a computer operating system can be explained by an increase in the number of facts added to its knowledge base. What our expert simulation knows is not different from what the intermediate or novice model knows - there is just more of it. Conversely, by deleting facts from an expert knowledge base, we are able to model novice and intermediate performance.

We have chosen to explore these issues in the UNIX domain because we wanted to study expert performance on tasks that experts actually do. As Holyoak (1989) points out, much of the expert-novice literature on problem solving revolves around having both experts and novices perform tasks that are instructional tasks for novices, but not necessarily tasks frequently performed by experts. For example, an expert physicist doesn't normally spend time solving the classical problems of physics, where a novice will do so in the context of an introductory class. It is of considerable interest how expert-novice differences will look if we investigate these differences in a task that both experts and novices perform in their everyday work.

UNIX is one exemplar, perhaps the best known, of an operating system. In general, the functions of an operating system go beyond what is apparent to the typical user. They may include, for example, controlling hardware and managing system resources as well as logging and managing files, or compiling and running programs. Functions like the latter, which are directly controlled by users through a software interface, are being modelled here, because they may come to be represented by the user in the form of knowledge that governs system and command use.

Two characteristics of the UNIX operating system are its modularity and its hierarchical organization. By modularity we mean that the system performs a number of discrete functions, only some of which may be of interest to any one user. The hierarchical nature of UNIX results in part from the "shell" interpreter, which allows users to interconnect UNIX modules and to control the flow of information between structures with "shell" commands. In fact, the fundamental assumption underlying the UNIX user interface is that complex commands can be created by concatenating simple commands with the use of advanced features that redirect command input and output (e.g., pipes). Doane and her colleagues (Doane, 1986; Doane, Pellegrino, & Klatzky, 1989, in press) have examined the use of these advanced features (e.g., input and output redirection) to determine how much training and experience is required before they can be put to effective use. Their empirical data provide us with the basis for our theoretical investigations.

In the experiments by Doane et al., subjects are given brief verbal instructions to produce legal UNIX commands, such as "Sort the first ten lines of the file TeaBag alphabetically." Our specific goals in this paper are (a) to characterize the knowledge that users of UNIX have that enables them to perform such tasks; (b) to simulate by means of the construction-integration model the emergence of a control structure, a script or plan, for the execution of the specified task, given a general knowledge base about UNIX and a particular task instruction; and (c) to account for empirically observed errors and failures by UNIX users of various levels of expertise by showing how missing items of knowledge can produce these results.

This third goal directly relates to questions about the nature of expertise. If we can model decrements in expertise merely by deleting facts from the knowledge base, then we have provided some evidence that the development of expertise can be characterized by a monotonic increase in the number of known facts as opposed to a qualitative shift in processing. Obviously, our efforts do not rule out the possibility that another model may be developed that accounts for these data in other ways; our goal is merely to determine whether our model can account for the data in a parsimonious fashion.

We start with a cognitive task analysis of knowledge necessary to perform UNIX commands. We then show how, according to the construction-integration theory, the right knowledge becomes activated and sequenced in response to a given instruction. Finally, we compare model predictions to experimental data to show how commonly observed errors arise. Specifically, we compare the model performance to that of human subjects performing the same tasks, UNIX command production for problems that vary in complexity, to evaluate how well the model accounted for the data. In summary, we propose to simulate user understanding of instructions and the generation of appropriate action plans by extending the discourse comprehension theory of van Dijk & Kintsch (1983), and Kintsch (1988). This extension is directly based upon the work of Mannes & Kintsch (1989), who, in a somewhat different context, showed that notions of discourse understanding could account for how people conceived action plans on the basis of their understanding of the instructions. Understanding an instruction means doing what is requested - not (just) being able to remember or summarize the instruction.

Mannes and Kintsch (1989) describe a continuum of action from automatic responses (like starting a car) to problem solving in new planning situations requiring deliberate thinking and planning. Their continuum can be effectively compared to that of Card, Moran and Newell (1983), where they describe the range in activities from routine cognitive skills to problem solving. In our research, we are modeling performance spanning this continuum. Some of the tasks, e.g. the production of single UNIX commands, fall at the automatic or routine end, where users either have the requisite knowledge or they don't. Other commands, ones that require the composition of a complex sequence-dependent string of single commands, fall toward the other end of the continuum. These tasks are difficult, even for the experts. To accomplish these tasks familiar single commands must be chained together in novel ways. Users cannot recall fixed scripts from memory, but must produce a new action plan for each task. The action plans for these tasks cannot be precompiled, because while the elements of some of the tasks may be familiar, and indeed overlearned, they are often put together in novel sequences. That is, we are not dealing with fixed scripts which can be retrieved from memory, but with emergent control structures that are generated in the context of a specific task.

The paper is structured as follows. First, we review the UNIX empirical data collected by Doane and her colleagues (Doane, 1986; Doane et al. 1989; in press). This is followed by a general description of the construction-integration model. Then we will describe the version of the construction-integration model that was developed for understanding UNIX commands. Following the discussion of this model, we discuss the specific simulations of understanding and producing various UNIX commands by users of different levels of expertise.

Performance Data

Doane et al. (1989; in press) examined the development of expertise within the UNIX operating system by studying UNIX users with varying levels of experience.

Included in these studies were both longitudinal and cross-sectional analyses of the emergence of command production expertise.

Subjects

Three groups of subjects differing in their expertise were used in these experiments; novices, intermediates, and experts. Subjects were assigned to one of these groups by Doane et al. (in press) based on two measures: Their responses to items on a pre-experimental questionnaire, and their ability to produce and comprehend legal UNIX commands. Subjects in these groups had, on the average, 8 months, 2 years, and 5 years of experience with the system, respectively. All of the subjects had taken a course which taught about pipes and other redirection symbols and required the use of these techniques for course homework.

Materials and Procedures

One part of the project was concerned with measuring users' performance in tasks requiring them to produce UNIX commands. Subjects were asked to produce the most efficient (i.e., requiring the least number of keystrokes) legal UNIX command that they could to accomplish a specified task. Tasks ranged in difficulty from individual, frequently used UNIX commands to composite commands that effected several actions that had to be sequenced appropriately using pipes or other input and output redirection symbols. The tasks were designed to assess the impact of two types of knowledge; knowledge of individual commands and knowledge of the processes involved in sequencing those commands properly. Tasks involving more elementary commands were designed to include elements that could be combined to complete composite commands. For example, the elementary commands **ls** and **lpr** can be combined using a pipe to create the composite command **ls|lpr**.

The total number of commands used in the production task was 93. These were divided into 3 blocks, with 31 terms in each block. Each block contained a third of each type of command (e.g., single, multiple, composite), and the presentation of the three blocks was counterbalanced across subjects. For each task, computer programs presented the instructions, gave practice trials, presented terms, and stored reaction times and answers. The task displayed a working directory and the current production request at the top of the screen throughout all trials. Subjects were tested individually in a session lasting approximately two hours. The second hour of the session was spent on a task that is not discussed here.

An example of a composite task would be "Display the first ten alphabetically arranged file names of the current directory". A component single would be "Display the file names of the current directory". A multiple would be "Display the file names of the current directory", "Arrange the contents of file TeaBag alphabetically on the screen", and "Display the first ten lines of file MilkShake". Thus, "single" commands required just one action, multiple commands required several independent actions, and composite commands required several actions that had to be sequenced appropriately using pipes and (or) redirection symbols. Multiple and composite commands were equal in length (in terms of number of keystrokes required to complete the commands), so that the essential difference between them was that composites had dependencies among their components whereas multiples did not.

Another type of command studied by Doane et al. were iterative single problems. These commands requested the user to repeat the same single command several times, such

as deleting several files. Iteratives could be implemented (optionally) by special syntax that allows the user to type in a single command only once, then have the system repeat the subcommand. To use the special syntax, the user types in a "wild card" character, which matches any file name meeting the specified constraints. For example, to perform the operation "**sort**" on File1, File2, and File3, the user might enter "**sort** File[1-3]". The system would repeat the **sort** command on any matching files (**sort** File1, **sort** File2, **sort** File3).

Results

The production data indicate that UNIX users differ markedly in performance, according to their history of use with the operating system. The striking aspect of these data was that only the experts could successfully produce the composites, even though the intermediates and novices could perform the other tasks that were designed to assess the component knowledge required to successfully generate a composite (e.g., singles).

Figure 1 shows the percentage of subjects at each level of expertise that correctly executed single commands (those singles which were also included in multiples), multiples, iteratives, and composites. Novices knew little more than half of the single commands, compared to about 90% for the experts. When single commands were strung together to form multiples, only a small performance decrement of about 5% was observed for the novices and intermediates, and practically none at all for the experts: If subjects knew a command, they could use it alone or together with other (unrelated) commands. The results for iteratives, however, are intriguing: One would think that if one knew a command, one could execute it just as well three times in a row as only once. But that was clearly not the case for novice and intermediate subjects. Composites, where there were dependencies between the commands, were by far the hardest. Neither novices nor intermediates showed much ability to execute composite commands, and even the experts got only about half of the composite commands right.

These results are somewhat surprising, since the knowledge necessary to perform the composites and iteratives is taught to intermediates and novices in their courses, and they must use the pipe to complete their homework assignments. They are taught that the pipe (|) is a symbol which redirects input and output between commands and that it is a feature of the UNIX interface which affords the construction of composite commands. A common use, as an example, is the command **ls|lpr**, which is constructed by concatenating the commands **ls** and **lpr** using the pipe symbol. Indeed, all of the subjects used a pipe at least once in the experiment, demonstrating their general familiarity with the concept. Nevertheless, they were unable to use pipes consistently in different tasks. The less expert groups have the knowledge, but can't use it productively in the sense discussed by Wertheimer (1945).

Doane et al. calculated the probabilities that a user was correct on complex iterative commands, given that he or she correctly produced the component single commands. For cases where subjects were correct on a single command, the percentage correct on the iterated version was calculated, yielding the conditional probability: correct on iterated single given correct on noniterated single. Similarly, the probability of correct on iterated multiple, given correct on the iterated version of its component singles, was calculated. A value of 1.0 would mean that performance on the coordinated command was predictable entirely from performance on its component singles. Finally, for cases where a subject had given a correct response to all of the components of a multiple when they were presented as single commands, the percentage of sufficient responses to the multiples as a whole was

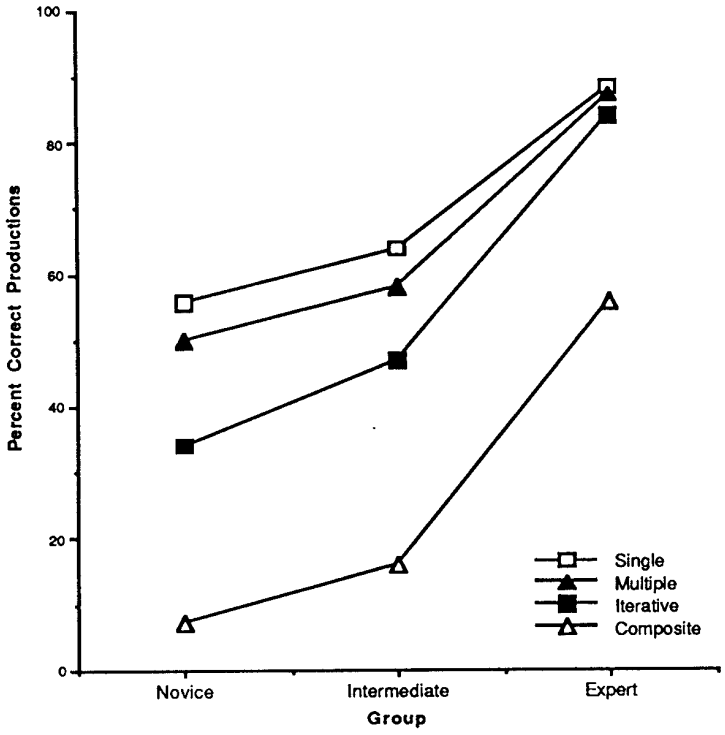


Figure 1. Percent correct productions by group and problem type. Adapted from Doane et al. (in press).

calculated. This produced the conditional probability: correct on multiple given correct on component singles.

As Figure 2 shows, this analysis indicated that performance on multiple commands was predictable from performance on the component singles, as long as iteration was not involved. Multiples with iteration, however, were not predicted from iterative singles for the lower-expertise groups, nor were singles with iteration predicted from the same singles without iteration.

Thus, iteration appears to add processing complexity above and beyond the component commands, for both singles and multiples. This seems surprising, since iterative commands could be accomplished by merely repeating a command without using special iterative symbols. To determine whether the difficulty with iteration reflects use of the iterative symbols or a more general phenomenon, Doane et al. broke the data down into iterative problems where the subject did or did not use the special symbol. Their results suggest that (1) experts use the symbol more often than intermediates or novices (percent symbol use of 13%, 18%, and 26%, respectively) and (2) all three groups were more likely to be correct on problems where the iterative symbol was used (89% vs. 72% for experts, 67% vs. 46% for intermediates, and 55% vs 28% for novices) and (3) the advantage for problems where a symbol was used increases as expertise decreases. Thus, it appears that the difficulty with iterative problems does not lie with the knowledge of the symbol per se, but reflects a more general conceptual problem.

To summarize the Doane et al. (in press) group data, novice and intermediate UNIX users may not know all UNIX commands, but if they do, they can successfully produce single and multiple commands. They cannot, however, put these known elements together using pipes and (or) other redirection symbols to produce the composite commands and to a lesser extent iteratives. As previously stated, the symbols that enable input and output redirection are fundamental design features of UNIX, and these features are taught in elementary classes. Doane et al. (in press) demonstrate that these features can only be used reliably after extensive experience (e.g. experts had on the average 5 years of experience with UNIX). It appears that there is a general conceptual problem for all groups. The goal of our simulation will be to identify the locus of this problem and its effects on command execution.

Doane et al. (in press) also examined a subset of their initial users two years later. Their longitudinal data demonstrated that most subjects increased in expertise. This increase was quite regular, in that individual growth profiles generally reflected the group data discussed earlier: subjects first acquired the ability to produce the simple commands before acquiring knowledge of complex commands.

Below we shall model the group and some of the individual production data of Doane et al. (in press). Specifically, we model intermediate and expert group production performance on a subset of their production tasks. We also model the individual performance of four users who were studied longitudinally. Prior to this discussion, however, we shall provide a general description of the construction-integration model.

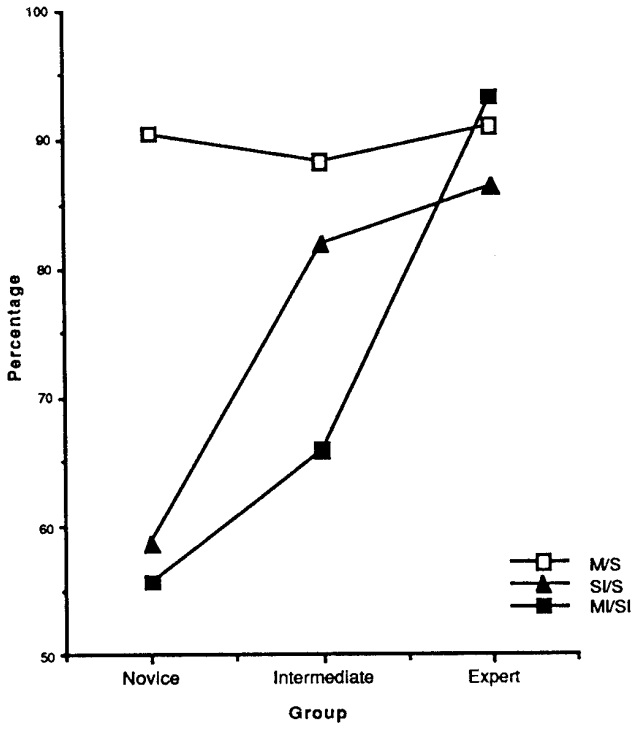


Figure 2. Percentage of coordinated commands correct, given correct component singles, by group and problem type (S=Single, M=multiple, I=with iteration). Adapted from Doane et al. (in press).

The Model

The Construction-Integration Model

The construction-integration model of Kintsch (1988) continues the development of a general theory of discourse comprehension, (e.g. Kintsch & van Dijk, 1978; van Dijk & Kintsch, 1983) by specifying how knowledge is used in the process of comprehension. In the construction-integration model comprehension is modelled as the construction of a semantic (propositional) representation of a discourse. In order to achieve a coherent representation, an integration process in the connectionist manner is employed; hence the name of the model.

This representation derives from two sources: first, the to-be-comprehended text itself, in the form of a propositional textbase, as in van Dijk & Kintsch (1983); second, knowledge activated by the text. Knowledge is represented in the model as an associative network of propositions and concepts - the comprehender's long-term-memory net. There are two ways in the model in which knowledge from this net can be added to the textbase. First, text propositions can select neighboring associated nodes from the long-term memory net, which will then be added to the textbase as elaborations. In the present application of the model, since we are dealing with very brief discourse, this phase of the process plays almost no role and will be neglected. Secondly, general task sets can activate relevant knowledge. For instance, if we know that we are reading an algebra word problem, the situationally relevant algebra knowledge will become part of the task representation; if we know we are supposed to execute UNIX commands, the situationally relevant UNIX knowledge will become part of the task representation.

How do people manage to activate just relevant knowledge and avoid being overwhelmed by endless irrelevancies? One common hypothesis is that the comprehension process is guided by such control structures as frames and scripts, so that only knowledge that fits into the currently dominant frame becomes activated. The construction-integration model proposes a different explanation. It activates knowledge without much guidance or concern for relevance, thus constructing a representation that includes much that is irrelevant and even contradictory. The representation it constructs is an interconnected network, however, so that it is possible to let this network settle into a state where nodes that do not belong are isolated, and contradictory nodes are rejected. A spreading activation process (i.e., the integration process), where propositions that belong together strengthen each other and isolated nodes become deactivated, achieves this goal.

The construction-integration model, therefore, has three essential components: the construction of a propositional textbase, the enrichment of that textbase through a process of relatively unguided, uncontrolled knowledge activation, and finally the establishment of a coherent, integrated representation consisting of both text and knowledge elements through a spreading activation process. The model, therefore, belongs to the group of "symbolic connectionist" models described by Holyoak (1989) which combine the advantages of a symbolic representation and production-system-like construction rules with the power of connectionist processing mechanisms.

Below we sketch in some more detail the essential features of this model, focussing on the Mannes & Kintsch (1989) formulation. We will then show how this general model of discourse comprehension can be adapted to simulate how UNIX users understand and execute instructions to produce commands.

Instructions are understood by forming a mental representation of the instruction itself elaborated with the rich knowledge maintained by expert users. We are concerned with three aspects of this mental representation: the elements from which it is put together; the relations among the elements that allow us to form a network of these elements; and the processes of integration that produce a coherent, contextually appropriate task representation from this network. This task representation is the model's understanding of the to-be-performed task, and it determines how the task will be executed.

The elements of the representation. The elements of the task representation are propositions, that is, conceptual units consisting of a predicate (often expressed linguistically as a verb, adjective, or connective) and several arguments (typically noun concepts or other propositions). Thus, propositions can be embedded within other propositions.

Propositions come from two sources, the text - the instructions, in this case - and the comprehender's knowledge -- in this case, what she or he knows about UNIX. In principle, text propositions are generated by production rules that take as their input a text and have as their output a set of propositions representing the meaning of that text. In practice, we bypass this stage and handcode a text into its propositional units (van Dijk & Kintsch, 1983). Knowledge propositions are also generated by production rules: there are productions that copy propositions from long-term memory, as well as productions that combine information provided by the text with information from long-term memory.

There are two kinds of knowledge propositions that we are concerned with here: general knowledge about UNIX, and knowledge about executing UNIX commands. Following Mannes and Kintsch (1989), the command knowledge of users tyakes the form of *plan elements*. Plan elements represent knowledge about the basic-level actions in UNIX, such as printing a file. An instruction is understood if the right plan elements have been selected, sequenced, and connected appropriately. Such a sequence of plan elements constitutes a *plan* which can be executed.

Plan elements in long-term memory contain as arguments generic terms such as "file". In a real task, we are dealing with specific files, say File A and File B. Hence, when a particular plan element is copied from long-term memory it must be bound to the actual objects that play a role in the task in question.

In order to understand an instruction, we must not only parse the instructional text into propositions and activate certain kinds of knowledge, but we also must know which propositions refer to facts in the world, which are items of knowledge, which are requests to do something, and so on. Hence we embed all propositional elements within epistemic predicates such as "this is a fact about the world", "this is a plan element", "this is general knowledge", "this is a request", and so on.

Tying the elements together into a network. Since the production rules that build propositions from the text as well as the rules that select knowledge from long-term memory are very general and unselective, many elements enter a task representation that should not be there. For instance, we simply select everything the system knows about UNIX and now must decide what is actually relevant to the task at hand. For this purpose, we first must interrelate all the elements we have created in a coherent network. Following van Dijk and Kintsch (1983), we use argument overlap among propositions and propositional embedding as one method to establish links among propositions. This is merely a default procedure, however, and, as in Mannes and Kintsch (1989), additional

rules must be formulated to generate appropriate links for certain special types of relations, primarily the causal relations among plan elements and the relations between facts in the world and plan elements. Both positive and negative (inhibitory) links will be created in this way.

In Figure 3 we summarize what we've constructed so far. We have a matrix that represents the task description, the general knowledge about UNIX, and the plan elements. All of the variables in this matrix are bound to the specific information introduced in the task description (i.e., to specific file names). We then integrate this matrix by spreading activation from the task description knowledge through the general knowledge to the plan elements.

Formulating a plan for action. For the task network generated in the manner sketched above, (see Figure 3), the spreading activation process determines which of the knowledge elements are relevant to the task at hand. This is accomplished by post-multiplying an initial activation vector (see Figure 3) by the constructed matrix, and then repeating multiplication until activation settles (further details are provided in a later section). Knowledge propositions and plan elements that are tightly related to the instructional text in the ways described above (e.g., argument overlap and propositional embedding) will become more strongly activated, while knowledge propositions that are unrelated or related via inhibitory links will tend to become deactivated, and this will be represented in the final activation vector values.

Once a coherent interpretation of the task network has been achieved, the system acts: it models the execution of the most highly activated plan element that can be executed, given the constraints of the present situation. Executing a plan element, however, has consequences: certain things happen in the world, changing the state of the world. These changes are modelled by adding whatever the outcome of the executed plan element was to the task network as one of the facts that is true in the world. Then, the whole process of integration and executing a plan element starts over again - until the final outcome implied by the task instructions has been achieved. At this point, the system has computed its plan: a certain sequence of plan elements that will do what the instructions requested. It is ready for real action.

Thus, our simulation first thinks about an action, computes what would happen if, then responds with another thought-action to the changed situation, and goes through several cycles of modeling what would happen until it generates the desired outcome. Then it does what it has planned. Planning in this system is, therefore, tightly bound to the existing state of the world. We always think only one step ahead, then take stock of the new situation we have created, and try again. There is no long chain of reasoning about subgoals, but a direct understanding of the situation, and keeping track of the changes in the situation that are brought about by the actions we are considering.

In the next section we shall describe in detail how what we have sketched above is realized in our model. Following that, we present our actual simulations of the Doane et al. data.

Construction of the Semantic Representation of UNIX Commands

The construction-integration model has been used to construct a simulation, called UNICOM, of how experienced and inexperienced users understand and execute UNIX commands. UNICOM knows how to use the basic UNIX commands. This knowledge is represented in the form of *plan elements*. Understanding a task instruction for this

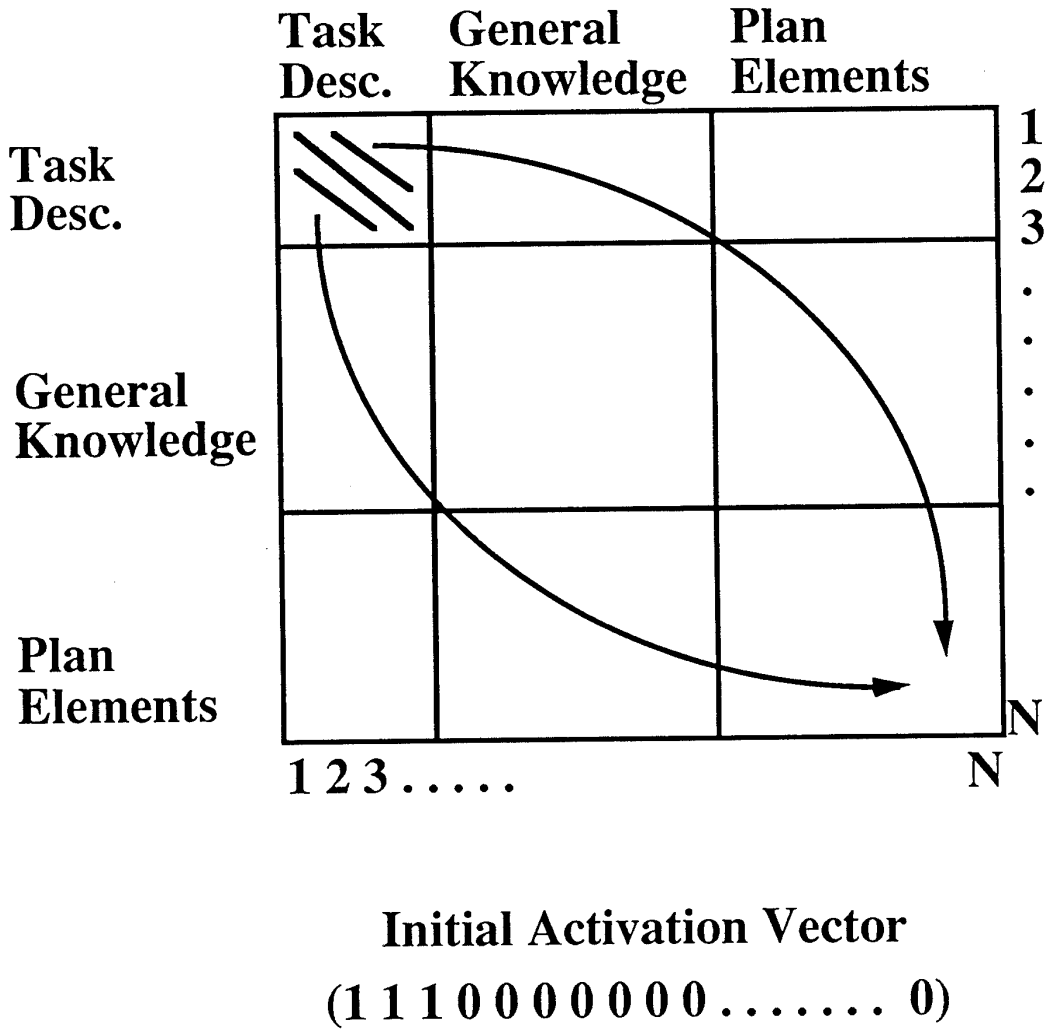


Figure 3. Summary of the construction and integration phases of the UNICOM construction-integration model.

simulation consists of selecting and sequencing plan elements so that when they are executed, the goals specified by the instructional text will be achieved. We call this forming a *plan*. A plan may consist either of a sequence of separate plan elements, or of composites formed from these plan elements by means of pipes, redirection, and iteration.

Comprehending an instruction involves understanding the instructional text itself and letting it retrieve the knowledge required to execute the instruction. For clarity of exposition, we shall distinguish between two types of knowledge, general knowledge about UNIX, and knowledge how to execute specific commands in UNIX - the plan elements.

Planning and action. Since we are concerned both with the planning and execution of commands, the course of the simulation for iteratives and composites can be described as two phases: it first understands the command, resulting in a plan for action that would achieve the desired outcome; then it executes this action plan by generating the actual command. For purposes of exposition, we will call this first phase the action-planning phase, and call the second the action-execution phase. UNICOM does not simulate the actual motor action of typing a command. Rather, it hands off the goal of typing a certain command or command sequence to an action module which works along principles described in Kieras and Polson (1985). This latter part of the system is beyond the scope of the present paper.

In the case of single or multiple commands, as soon as an appropriate plan element has been selected, the system is in the action-execution phase, and that element is executed. Single problems have one outcome, and this outcome is immediately selected. Multiple problems have several outcomes, but once again each one is achievable by a single plan element, and UNICOM enters the action-execution phase as soon as such an element is found, skipping the action-planning phase. Hence the distinction between planning and action plays no role in these cases. As soon as we know how to achieve a desired outcome, we act.

The situation is different for composite and iterative problems, however. The outcome implied by these problems is the result of a string of coordinated actions. Hence the system is in an action-planning phase until the system has found all the plan elements necessary to arrive at the requested final outcome. The system does not stop at the end of an action-planning phase. We have included a control structure which makes the system continue into an action-execution phase once an action-planning phase is completed. Thus, the system first plans and then executes the completion of composite and iterative problems. By delaying the action-execution phase until a complete plan has been constructed, we allow UNICOM to figure out, for instance, that it needs to do A first, then B using the outcome of A as the input to B, and that this could be best achieved by constructing a pipe A|B. The situation for iteratives is analogous, as will be shown below, when the actual mechanism UNICOM uses for building pipes and iteratives will be discussed in detail.

Separating the action from planning requires that the system will have to keep track of what it did during the action-planning phase. UNICOM does this via trace propositions of the form TRACE[COMMAND1^A COMMAND2^B], where COMMAND1 is a specific command that is selected first for execution and is bound to the file A (noted by ^A) and COMMAND2 is a specific command which is selected for execution second and is bound to the file B. In addition, files A and B are specific files included in the request and outcome propositions.

The instructional text. The subjects in Doane et al. (1989) received instructions in verbal form, e.g. (1) "Display the first ten lines of the alphabetically arranged contents of the current directory on the screen."

The simulation received this text in propositionalized form, after van Dijk and Kintsch (1983). For (1), there would be three propositions, two existential proposition asserting that there was a directory and there were files in the world, and a (complex) proposition specifying a request. Stripped of the actual formalism used in the simulation, we obtain as the propositional representation of (1)

(2) INTHEWORLD[DIRECTORY]

INTHEWORLD[FILES]

REQUEST[DISPLAY FIRST-TEN-LINES DIRECTORY

ALPHABETICALLY-ARRANGED ON-SCREEN]

Note that (2) contains two pieces of information not explicit in (1): we have made an inference that there are files listed in the directory, and we are marking the epistomological status of each proposition, such as INTHEWORLD or REQUEST.

Upon encountering a proposition marked REQUEST, UNICOM searches its long-term memory store for the outcome of the requested action. Specifically, it looks for a proposition marked OUTCOME that is associated with the particular request received. If it does not find an outcome, it cannot understand the instruction. In the present case, it would find

(3) OUTCOME[REQUEST[DISPLAY X ON-SCREEN],

INTHEWORLD[X ON-SCREEN]]

where X is the string {FIRST-TEN-LINES DIRECTORY- ALPHABETICALLY-ARRANGED}. (3) provides the model with crucial information about what knowledge to activate as well as when to stop planning. (2) and (3) together form the core of the textbase upon which all further computations are based.

General knowledge about UNIX. From a cognitive task analysis of the knowledge required to successfully produce composite commands, we developed a UNIX fact base containing 66 propositions. The knowledge to successfully produce composite commands includes knowledge of the commands themselves, knowledge in general that input and output can be redirected, and specific knowledge about input and output redirection as it relates to each command. Some examples of general knowledge about UNIX in UNICOM's long-term memory are

(4) KNOW[REDIRECT INPUT OUTPUT]

KNOW[REDIRECT INPUT TO-COMMAND FROM-COMMAND]

KNOW[REDIRECT INPUT TO-FILE FROM-COMMAND]

KNOW[REDIRECT COMMAND INPUT OUTPUT PIPE]

KNOW[REDIRECT INPUT TO-FILE FROM-COMMAND

FILTER]

Once again, we are not showing the actual LISP expressions used in the simulation, but employ a simplified notation for clarity of exposition. UNICOM knows 14 general knowledge propositions. In addition, there are 52 specific knowledge propositions, relating to specific UNIX commands, such as

(5) KNOW[**ls**]

KNOW[REDIRECT OUTPUT FROM-**ls** TO-**command**]

KNOW[REDIRECT OUTPUT FROM-**ls** TO-FILE]

for the command **ls** (list contents of a directory). For the ideal expert the UNIX fact base contains the knowledge sufficient to successfully produce all of the UNIX commands simulated. For simulations of the intermediate group, as well as for simulation of individuals, facts were systematically deleted from UNICOM's long-term memory based on an analysis of group and individual performance data, respectively. This procedure will be detailed below in the section describing specific simulations.

Plan elements. The third type of knowledge that UNICOM has is planning knowledge, the knowledge of the commands necessary to accomplish the requested task. Plan elements are the basic planning (and action) units that the system has available. Each plan element is derived from and corresponds to a "single" command, such as List-a-directory, Alphabetically-arrange-a-file, Print-a-file, etc.

The decision to have plan elements correspond to the single commands is based on two observations. First, single commands correspond to the what can be considered as the basic level of categorization of the UNIX system (e.g., Rosch, 1975). As such, it makes intuitive sense to us to propose them as separate plan elements in the model. The second reason to represent the singles as separate plan elements is because the production data collected by Doane et al. (in press) provides information about user knowledge of single commands. Thus we can easily determine whether a particular user knows how to execute a single command (i.e., has general knowledge about a command and has a single plan element) by scoring their production data.

Plan elements are represented in an extended propositional format, where each plan has three arguments: a plan name (e.g., ALPHABETICALLY-ARRANGE CONTENTS-OF-FILE), a set of preconditions which must exist in the world for the plan to be executed (e.g., the file to be sorted must exist), and a set of outcomes of the execution of the plan element (e.g., the sorted contents of the file exist). For the sake of brevity, we will use the command name rather than the name of the plan element to refer to a plan element. Thus, the plan element corresponding to **sort** is represented as

(6) (a) PLANELEMENT[ALPHABETICALLY-ARRANGE
CONTENTS-OF-FILE^$]

(b) INTHEWORLD[FILE^$, LOCATION FILE^$]
[KNOW **sort**]

(c) INTHEWORLD[ALPHABETICALLY-ARRANGED
CONTENTS OF-FILE^$]

where FILE^$ should be read as "any file". (6a) gives the name of the plan element, (6b) its preconditions (some file exists, and we know its location), the UNIX command **sort** is known, and (6c) its consequences (the contents of the file will be sorted).

Plans. Plans are distinct from plan elements. They consist either of the sequence of selected plan elements, or of these plan elements tied together in a composite plan, using a pipe or other redirection command. Plans are selected for execution only when their preconditions are satisfied. One of their preconditions requires that an action-planning trace exists which will satisfy the request and outcome specified in the task description. During its action-planning phase, UNICOM keeps a trace of the plan elements it has selected. As it creates the action plan to produce the desired outcome, it includes this trace in the composite plans. Thus, at the end of an action-planning phase, the appropriate composite plan should have a trace of the correct action-planning sequence, and this will allow it to be selected for execution.

The composite plans are structured just as plan elements, with names, preconditions, and outcomes. The plan to build a pipe is represented as

(7) Name:

PLAN[BUILD-PIPE COMMAND1^$, COMMAND2^$]

Preconditions:

KNOW[REDIRECT[OUTPUT, COMMAND1^$],

[INPUT,COMMAND2^$]],

TRACE[COMMAND1^$[OUTPUT^X]

COMMAND2^$[INPUT^X]]

Consequence:

OUTCOME[COMMAND1^$ | COMMAND2^$]

where COMMAND1^$ is any command that has OUTPUT X and has been selected first in the action-planning phase, and COMMAND2^$ is any command that has been selected second in the action-planning phase and that has input X. The first precondition simply says that the user knows that output from some specific command COMMAND1^$ can be redirected to another command COMMAND2^$. The second precondition is fulfilled when COMMAND1^$ and COMMAND2^$ have been selected in the planning process for a particular task. In other words, if it knows about redirection between two commands, and a trace exists of having previously selected these commands in the proper order, UNICOM constructs the requested piped command sequence. That is, selecting this plan for execution will cause the execution of a particular pipe command, tying together the selected plan elements. BUILD-REDIRECT and BUILD-ITERATE plans are generated analogously. A more detailed description of how composites actually work in UNICOM is given below where we discuss simulating the less expert user.

The representation of the BUILD plans are based on cognitive task analyses of how users are constructing their composite productions. They are not directly based on empirical data, since we simply do not have empirical data at this point to provide direct evidence for these representations. Our current research efforts are focussed on exploring this issue further.

The plans to build either pipes, redirection, or iterative commands will only be selected for execution if their preconditions are met. They will only be met in the context of composite and iterative problems, and they will not be met in the context of singles and multiples. For example, a single command will never satisfy the precondition for a BUILD-PIPE plan: A single will never include the requisite second command that will take the first commands output as input (see 7 above).

Variable binding. UNICOM has a general knowledge base or long-term memory, as described above, which is the same for all tasks. When UNICOM is given instructions to perform a specific task, it sets up a *task network* that is specific for this particular task. The task network consists of the propositional representation of the instructional text (for (1), this would be (2) and (3)), and all the knowledge that UNICOM has about UNIX (that is, propositions as in (4)-(7)).

Some of the plan elements, e.g. (6), contain variables, however. The plan element to alphabetically arrange the contents of a file can be applied to any file, but in a given task there may exist specific files, say the files A and B. All variables in the plan elements have to be bound to objects actually found INTHEWORLD or specified by the OUTCOME proposition. If there are two files specified in a problem, then we obtain two plan elements, say [ALPHABETICALLY-ARRANGE CONTENT OF-FILE^A] and

[ALPHABETICALLY-ARRANGE CONTENTS OF-FILE^B]; if no files exist, no plan to alphabetically arrange the contents of a file will be included in the task network.

Thus, the task network consists of a variable number of items, depending on how many of the variables in the plan elements can be bound to objects in a specific situation.

The task network. The elements of the task network are inter-related according to the following rules.

(a) An outcome proposition in the textbase is connected to all plan elements with the same outcome; in the simulations reported below, a numerical value of +1.5 was used for this type of connection.[3]

(b) Inhibitory connections are set up between INTHEWORLD propositions and plan elements with identical outcomes; a numerical value of -10 was used for this type of connection. Thus, plan elements that would produce an outcome that already exists in the world are prevented from being selected.

(c) A third type of connection exists between plan elements; these connections embody the causal knowledge of UNICOM. If a plan element requires a certain condition to exist in its list of preconditions, then this plan element is given an excitatory link to the plan element whose outcome satisfies that precondition. An example of this relationship is shown in Figure 4. Conversely, plan elements are given an inhibitory link to all plan elements that destroy one of their preconditions. An example of this relationship is also shown in Figure 4. This pattern of excitatory and inhibitory links among plan elements represents the user's knowledge about the causal relationships among plan elements. Numerical values of +.7 and -10 were used for the causal links among plan elements.

(d) All other relationships among the elements of the task network are approximated by a default rule: propositions that share a common argument are connected by a value of +.4. Propositional embedding is treated just like argument overlap.

Thus, two propositions P1 and P2 (these may be text propositions, or any type of knowledge propositions) are connected by a value of 2*(.4) if they have 2 arguments in common, or .4 if P1 is an argument of P2.

Argument overlap is used here to approximate the manifold and rich connections between knowledge propositions that exist in the minds of real users - ranging from semantic relations to contextual associations. There can be no question that argument overlap provides but a crude approximation of these relations, but it is simple and objective, and it has proven workable, if far from ideal, in other research on text processing (e.g., van Dijk & Kintsch, 1983). Note that most connections in the task network are symmetric, except for the causal chains and all inhibitory connections.

The construction rules of the model thus generate a task network consisting of (a) the text propositions, including a REQUEST; (b) the OUTCOME of that REQUEST, as

---

[3]No attempt was made to find "best" estimates of connection strengths; we simply copied these values from Mannes & Kintsch (submitted).

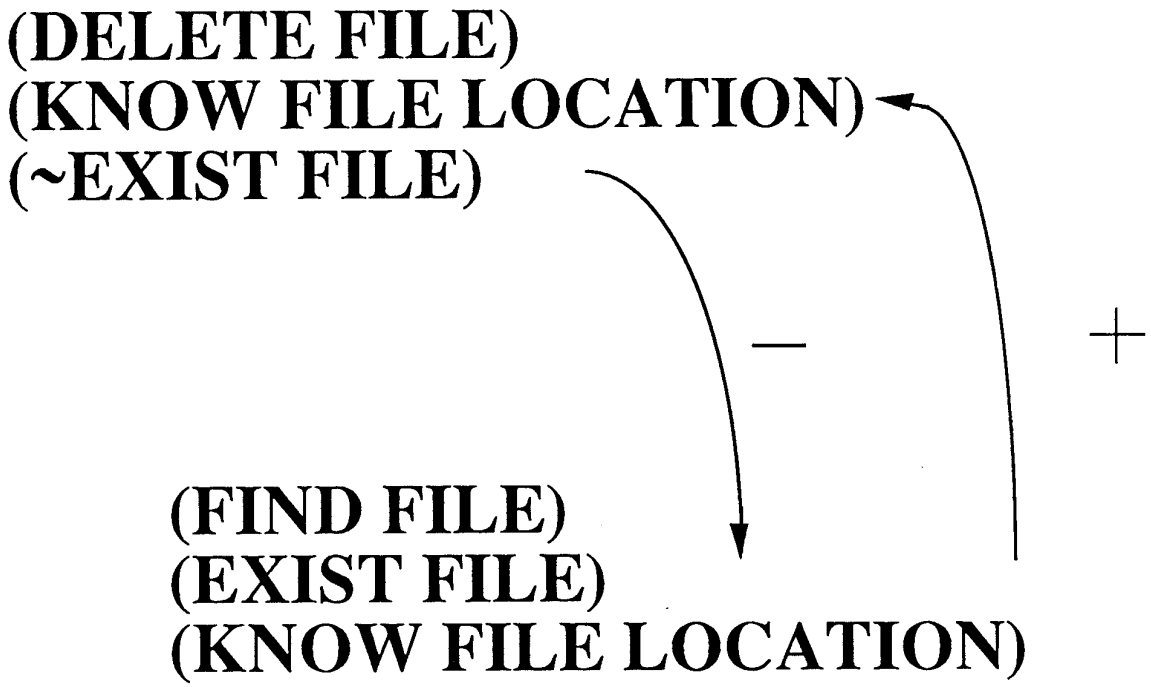


Figure 4. Summary of relationships between plan elements.

retrieved from long-term memory;[4] and (c) all the knowledge about UNIX that the system has - general knowledge propositions, propositions specifying the syntactic properties of specific commands, and finally the commands themselves in the form of plan elements (see Figure 3.). Variables in the plan elements are bound to objects specified in the textbase, thus sometimes creating duplicate plan elements, one for this object, and one for that. Finally, all the elements in the task network are interrelated. For the most part, these relationships are based on the overlap of arguments, except for certain relationships which are particularly important in the present context, involving causal links among the plan elements and the links between plan elements and the real and desired state of the world.

Integration

The weak construction rules employed here have generated a task network that contains all situationally possible plan elements. To come up with a sensible plan, we need to select among these unwanted riches. The integration process does exactly that by activating those plan elements that fit into the task context as established by the instructional text, and by deactivating irrelevant and (or) contradictory plan elements.

The task network is represented as a coherence matrix, with the propositions that have been generated above as the row and column elements, and their interconnection strengths as the cell values of the matrix (see Figure 3.). In addition, we define an activation vector, specifying for each element in the task net its activation value. Initially, this activation value is set to 1 for the text propositions and the retrieved OUTCOME proposition, and 0 for all the knowledge elements (see Figure 3.). Using this starting vector, the activation is propagated throughout the task matrix until a final activation vector is obtained. Specifically, the activation vector is multiplied by the coherence matrix until the activation in the network settles, i.e., when the average change in vector activations between iterations is less than .0001. After each multiplication, elements with negative activation levels are reset to 0, all elements corresponding to INTHEWORLD propositions are reset to 1 to trap their activation, and all remaining activation values are normalized such that they sum to 1 (Rumelhart & McClelland, 1986). The pattern of activation values in the final vector determines the plan element that the system will wish to select for execution first. A simple decision rule is used to select plan elements for execution: the most activated plan element whose preconditions are met INTHEWORLD is selected.

Integration cycles. The system models the production of commands of any complexity by a sequence of construction-integration cycles. A task network is constructed, it is integrated, a plan element is selected for execution, and its outcome(s) change the state of the world. Thus, a new task network is obtained. It is integrated, another plan element is selected for execution and its outcome changes the state of the world, giving rise to yet another cycle of construction and integration. The model continues processing in cycles until the desired state of the world is achieved (the originally retrieved outcome of the requested task now exists in the world), or until the system fails.

When some plan element is selected for execution, it will produce a certain outcome. This outcome is represented by one or more propositions which are specified in

---

[4]In the actual simulation, this step is hand-coded.

the outcome field of that plan element (see Figure 4). Executing the plan element will change the state of the world, specifically, it will add the outcome(s) of the executed plan element to the network as INTHEWORLD facts. These new propositions are connected to the existing task network using the same rules as before, but in a new construction phase. One major result of this step is that the plan element whose outcome has been added to the in the world fact set will now be inhibited, because its outcome now exists INTHEWORLD.

UNICOM, therefore, does not just plan in some formal problem space. Its comprehension processes are contextually situated in a dynamically changing world. UNICOM models what happens in the world, and updates its model continually. It tries to respond to a given situation in terms of its goal, then considers a new, changed situation, and so on, until its goal is realized.

Selection Traces. Recall that in the action-planning phase, UNICOM has not done anything, but merely modelled what would happen if it did certain things in the action-execution phase. Once UNICOM has convinced itself that if it followed a certain course of action, the desired outcome would result, it actually engages in those actions in the action-execution phase. For this purpose, it must keep track of the plan elements that it had executed during the action-planning phase. It does this by forming or updating a trace proposition whenever it selects some plan element for execution. In general, this trace has the form COMMAND1^A COMMAND2^B etc., where the first variable may be bound to **head** and the second to **sort**, yielding for example TRACE[**head**^A **sort**^B].

The existence of such a trace may, however, change the task net by enabling a plan like (7) to be selected for execution. Therefore, a new integration needs to be performed. As a result, the composite plan will now emerge with the highest activation (it is more richly connected than single plan elements), and it, rather than the sequence of single plan elements, will become the basis for action. Thus, UNICOM does what the subjects were instructed to do: use a composite plan such as a pipe whenever possible, and whenever it knows how to.

Accounting for Errors

UNICOM's knowledge base for the simulation of the prototypical expert include all facts required to successfully produce all of the commands simulated, so it will not make errors. In order to account for novice and intermediate performance, we need to provide a systematic explanation of their errors in the context of the theory. We make two major assumptions about the knowledge and the performance of less expert users. First, we make the assumption that such users have deficient knowledge and we represent these deficiencies by lesioning UNICOM's knowledge base. Two kinds of knowledge deficiencies are considered. First, users may not know all the single command plan elements. In this case the model would not have all of the single plan elements and it may be missing some of the general knowledge about a command (e.g., knowing that **sort** is a command). Second, their knowledge about the various redirection properties may be incomplete. In this case the model will not have all of the plans (e.g., BUILD-PIPE), and may be missing some of the general UNIX knowledge about redirection (e.g., knowing that input and output can be redirected).

The second assumption is that less expert users have trouble keeping track of the traces of their action-planning phase. This performance deficiency may reflect either a lack of knowledge or a short-term memory problem: less expert users simply fail to build and (or) retain sufficiently detailed action traces during action-planning.

In our simulations of intermediate and novice users we have not added misconceptions to the data base in the sense of Buggy (Brown & Van Lehn, 1980), but only deleted various facts or strategies from the perfect expert repertoire. Surely users have misconceptions, but our simulations merely explore the consequences of missing knowledge. In addition we do not simulate the resource constraints that would lead to various kinds of short-term memory failures.

Simulations

In this section we will describe some of our actual simulations of how UNIX commands are understood. We begin by describing the knowledge required to simulate the prototypical expert producing single, multiple, composite, and iterative commands. We then discuss the simulation of the intermediate group producing a composite and an iterative command sequence. Finally, we will discuss the acquisition of expertise in UNIX in the context of four simulations of individual users.

The subjects in Doane et al. (1989) were told to execute the UNIX commands they were given using pipes, redirection and iteration whenever possible. They were also told that there was a directory, and a list of file names. They were instructed to respond with "unknown" when they did not know what to do. These general pragmatic constraints remain the same for all the specific tasks UNICOM is given. Propositions specifying this task environment - the same for all tasks - are a part of all task representations that UNICOM constructs. In addition to these common elements, each new task contains a specific REQUEST, as well as a corresponding OUTCOME, which is usually not specified by the instructional text but must be retrieved from long-term memory, using the REQUEST as a retrieval cue.

Expert Performance

For the simulation of expert performance, the UNIX fact base consists of all the knowledge required to successfully produce composite commands. Specifically, the knowledge base contains knowledge of the commands simulated, knowledge in general that input and output can be redirected, and specific knowledge about input and output redirection as it relates to each command.

Simulation of the prototypical expert performance required the use of nine plan elements and three composite and iterative plan elements. The nine plan elements correspond to the nine single commands that were simulated (**ls**, **sort**, **head**, **tail**, **cat**, **more**, **lpr**, **nroff**, **wc**). The remaining three plans deal with composites and iteratives. Two of these are used in the case of composite problems: BUILD-PIPE and BUILD-REDIRECT. As we have seen in (7), BUILD-PIPE has two kinds of prerequisites. Certain general knowledge propositions marked as KNOW must be part of the task net - that is, our simulated user must know about pipes and their properties. Secondly, a trace of appropriate actions must be present.

Finally, the simulations contain a plan called UNKNOWN, corresponding to the instructions given to the subjects to respond "unknown" when they could not solve a problem. This plan will be selected for execution when no other plan or plan element whose outcome is related to the request can be selected for execution.

Singles. We chose as our example the **ls** command, but the solution to all of the other single problems are similar. The command **ls**[5] lists the contents of the current directory on the user screen. The specific request being simulated was stated in English as "Display the file names in the current directory on the screen". Thus, this request and outcome were propositionalized as follows:

REQUEST [DISPLAY DIRECTORY FILE-NAMES ON-SCREEN]

OUTCOME [DIRECTORY FILE-NAMES ON-SCREEN)

The FILE^$ variables in all plan elements were bound to DIRECTORY. Since only one file name was mentioned in the task instructions, no plan elements were duplicated. Rather, the simulation of this task included just 9 plan elements whose file-name variables were bound to the name of the current directory. The task network was created using the rules previously described to relate the elements within the task network.

The relationships between all pairs of text propositions, the propositions representing facts about UNIX, and the plan elements formed a 88 x 88 matrix for the task network. Numerical link strengths were assigned to the task network as described above.

A 1 x 88 initial activation vector was set to 1 for the 10 task description elements, and 0 for the remaining elements. The activation vector was multiplied by the task network until the average change in activation values was less than .0001, requiring nine iterations. For the sake of brevity, we will use the command name rather than the name of the plan element to describe the sequence of plan element activations. The plan element corresponding to the **ls** command turns out to be the most activated plan element (because of its connection with the OUTCOME proposition), its preconditions are met, and it is executed in the action-execution phase of UNICOM. Figure 5 shows the final activation values of the 9 plan elements: only **ls** is strongly activated (see Figure 5.), though there is some leakage to other plan elements. The **ls** plan element is most activated due to the overlap between its outcome and the desired outcome. Some of the other plan elements (e.g., **head**) receive some activation because they also have the term "display" in their outcome fields, and consequently, they partially overlap with the desired outcome.

---

[5]We are simulating the use of UNIX where the default **ls** does not alphabetically arrange the listing of the file names.

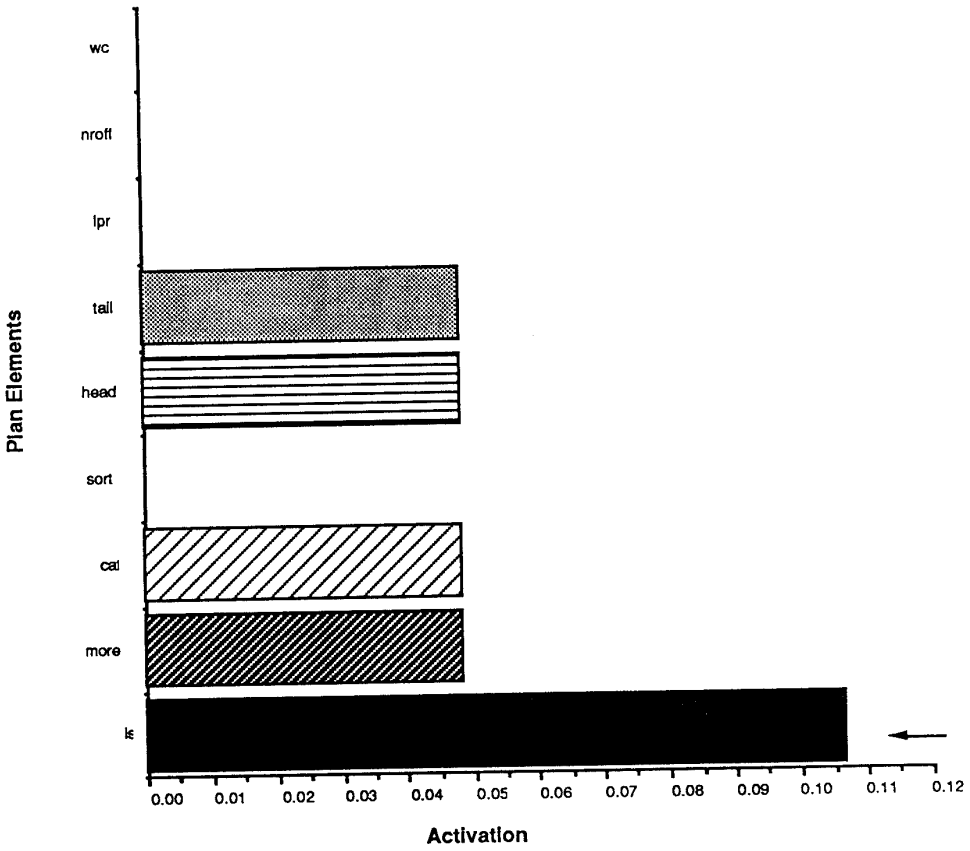


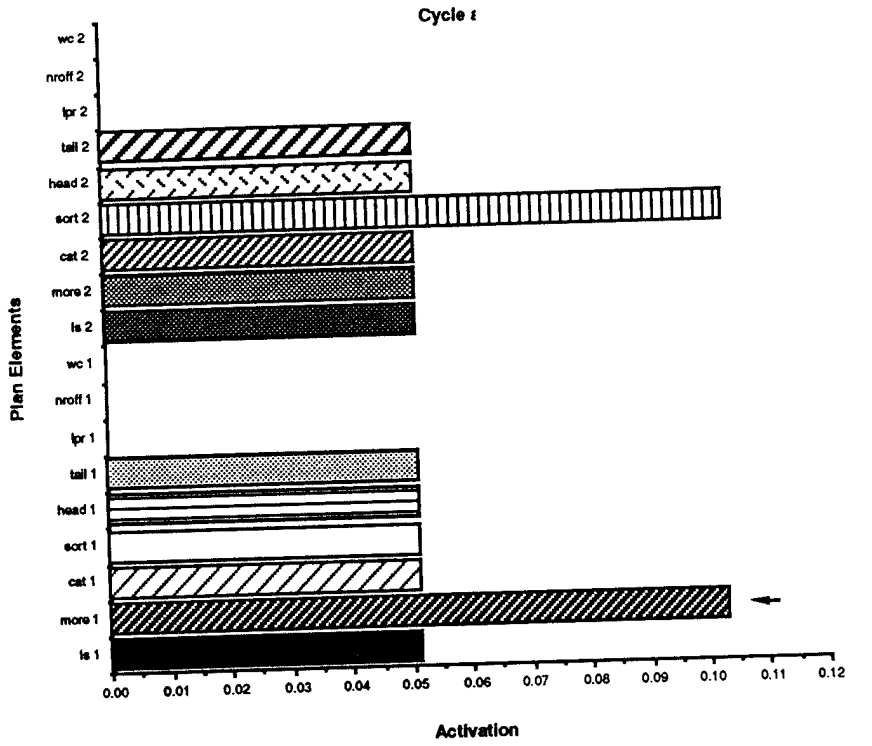
Figure 5. Activation values of plan elements from the UNICOM simulation of the single ls task.

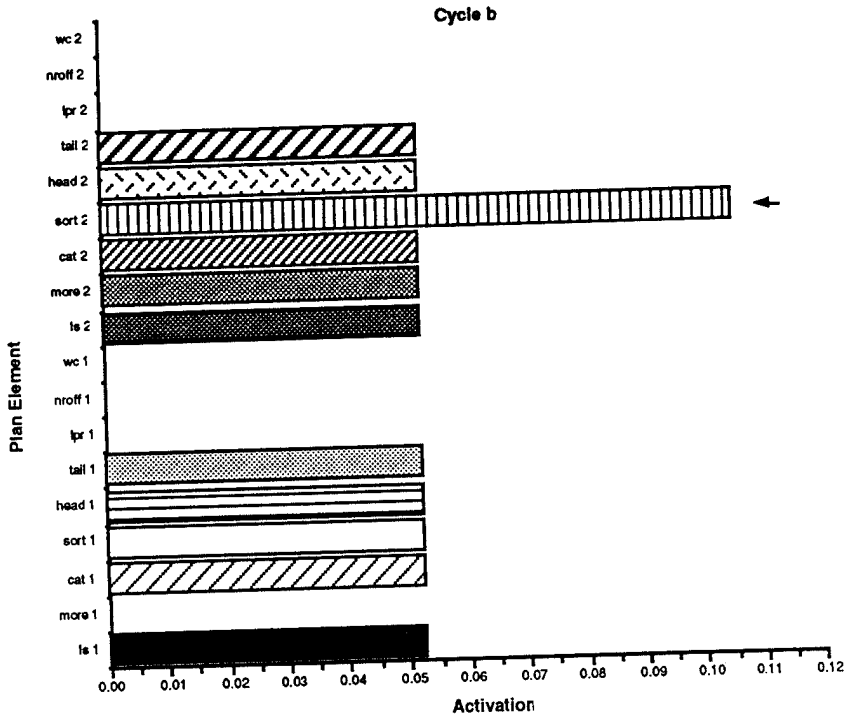
Multiples. The second task to be described is the simulation of a multiple command. A multiple is just a group of single commands. There are no interactions between the various commands that make up a multiple command, and UNICOM performs a multiple command in exactly the same way as it would perform the sequence of the corresponding single tasks (i.e., requiring only an action-execution phase), except that one large task net is created rather than several separate ones. Figures 6a and 6b show the final activation values for the task "Display the contents of the file A, and alphabetically arrange the contents of the file B": both **more**[A] and **sort**[B] are equally activated , the **more** command is arbitrarily chosen for execution first (see the arrow in Figure 6a.), and on the next cycle **sort** is executed (see the arrow in Figure 6b).

Composites. The composite problem which we have used as an example before is the request to produce **ls|sort|head**. The actual text is given in (1). The corresponding textbase is shown in (2) and (3). Initially, the plan elements are bound to the file DIRECTORY (plan elements numbered 1 in Figures 7 a-d) and to the file DIRECTORY-ALPHABETICALLY-ARRANGED (plan elements numbered 2 in Figures 7 a-d). These are the two files specified in the task net, the former by the instructions as existing in the world, the latter by the implied outcome proposition as part of the desired outcome. Note that for human subjects, recognizing these interactions is a non-trivial task. It requires parsing the instruction into the component commands and their interrelated input and output contents.

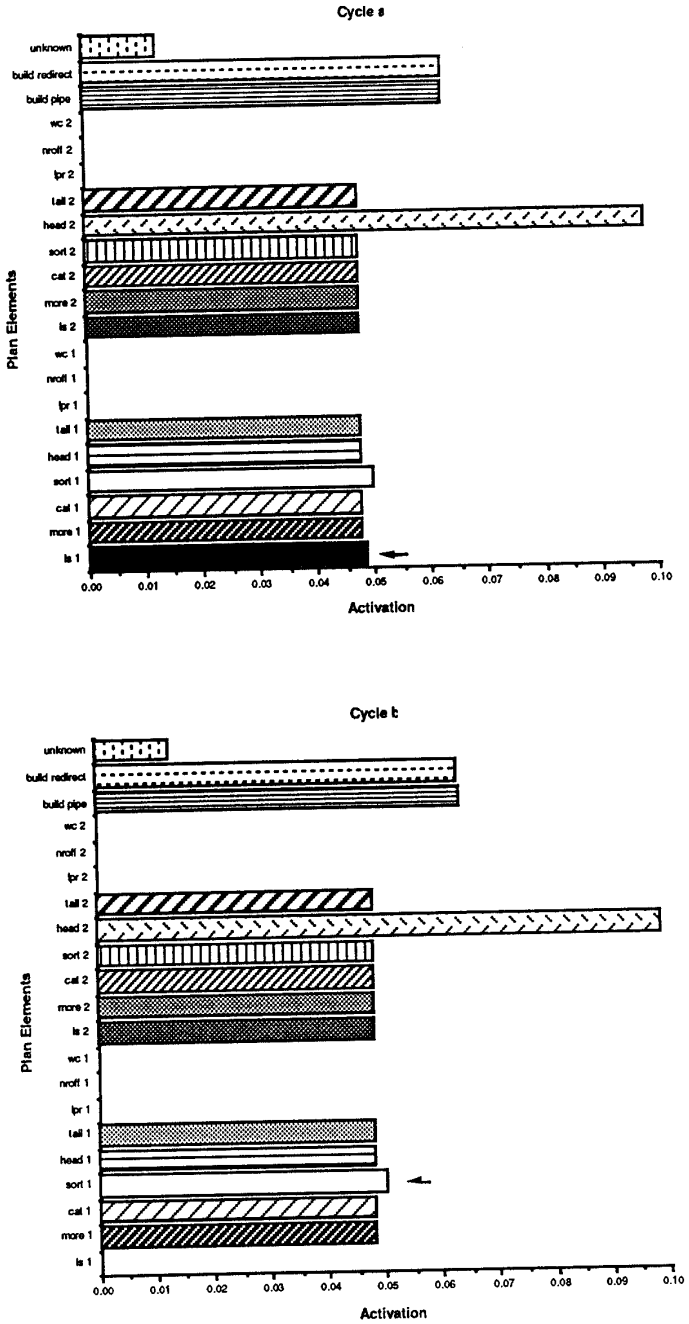
Figure 7a shows the activation values of these plan elements after the first processing cycle. The system wants to execute the plan elements corresponding to **head** and **sort**, but can't. The **head** plan element requires that the DIRECTORY-ALPHABETICALLY-ARRANGED exist INTHEWORLD, and **sort** requires that the DISPLAY FILE^DIRECTORY exist INTHEWORLD, and neither of these preconditions are met. The next most highly activated plan element is **ls**, however, which is selected for execution. This changes the task net in two ways: a proposition INTHEWORLD[DISPLAY FILE^DIRECTORY] is added to the task net, and the trace COMMAND1^**ls** is added to all of the composite plans.

The activation values of the plan elements in the second processing cycle are shown in Figure 7b: again **head** is strongest, but it cannot be executed; we have, however, created the preconditions for **sort** in Cycle 1, hence **sort** is selected for execution. It creates INTHEWORLD[FILE^DIRECTORY-ALPHABETICALLY-ARRANGED] and a record of its firing is added to the trace. On the third processing cycle (Figure 7c) one of the plan elements bound to FILE^DIRECTORY-ALPHABETICALLY-ARRANGED is selected for execution. There are several consequences: INTHEWORLD[FIRST-TEN-LINES DIRECTORY-ALPHABETICALLY-ARRANGED ON-SCREEN] is added, which matches the requested OUTCOME (3), telling UNICOM that it has achieved its planning goal; at the same time the firing of **head** is added as a third component to the trace proposition, which now becomes TRACE[COMMAND1^**ls** COMMAND2^**sort** COMMAND3^**head**]

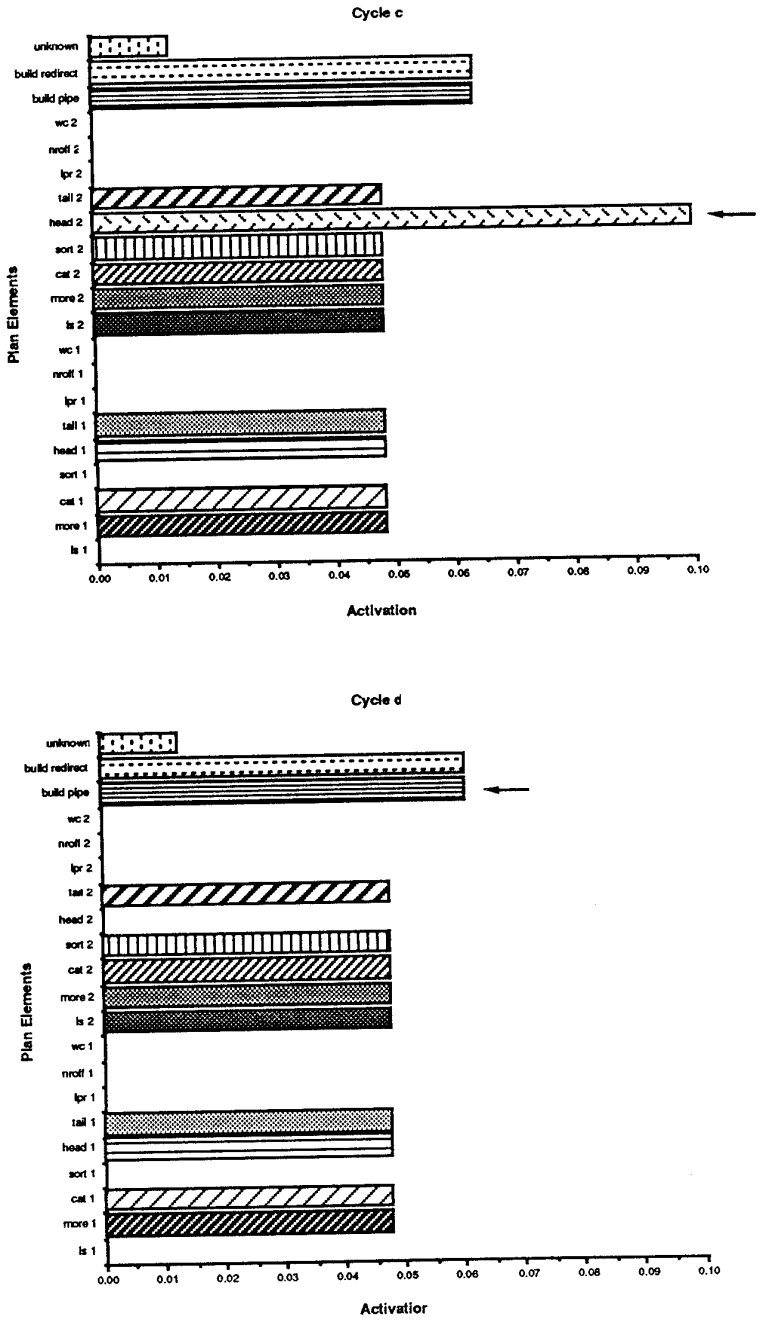




Figures 6a and 6b. Activation values of plan elements for cycles a and b from the UNICOM simulation of the **more** and **sort** multiple task.



Figures 7a and 7b. Activation values of plan elements and plans for cycles a and b from the UNICOM simulation of the **ls|sort|head** task.



Figures 7c and 7d. Activation values of plan elements and plans for cycles c and d from the UNICOM simulation of the **ls|sort|head** task.

The model could now enter into its action mode, and execute the three plan elements it has selected (and kept track of in the trace). However, the existence of the trace proposition now enables the selection of the BUILD-PIPE plan for execution, and it will construct a piped command. To select this plan, UNICOM enters its fourth processing cycle, the outcome of which is shown in Figure 7d: The BUILD-PIPE plan which in this context produces the piped command [COMMAND1^**ls**| COMMAND2^**sort**|COMMAND3^**head**] gathers the largest amount of activation, and it, rather than its three separate components, becomes the basis for action. We are modelling expert behavior here, so that all the knowledge prerequisites for building a pipe are fulfilled. As previously stated, this knowledge includes facts about redirection in general, knowledge of the pipe symbol, and knowledge of redirection using the pipe specific to each command.

BUILD-PIPE rather than BUILD-REDIRECT is slightly favored because the task net contains more propositions about pipes than redirection. That is, in the instructions and in the knowledge base, there are more propositions whose arguments overlap with those of the BUILD-PIPE plan. Thus, more activation is funnelled to the pipe plan.

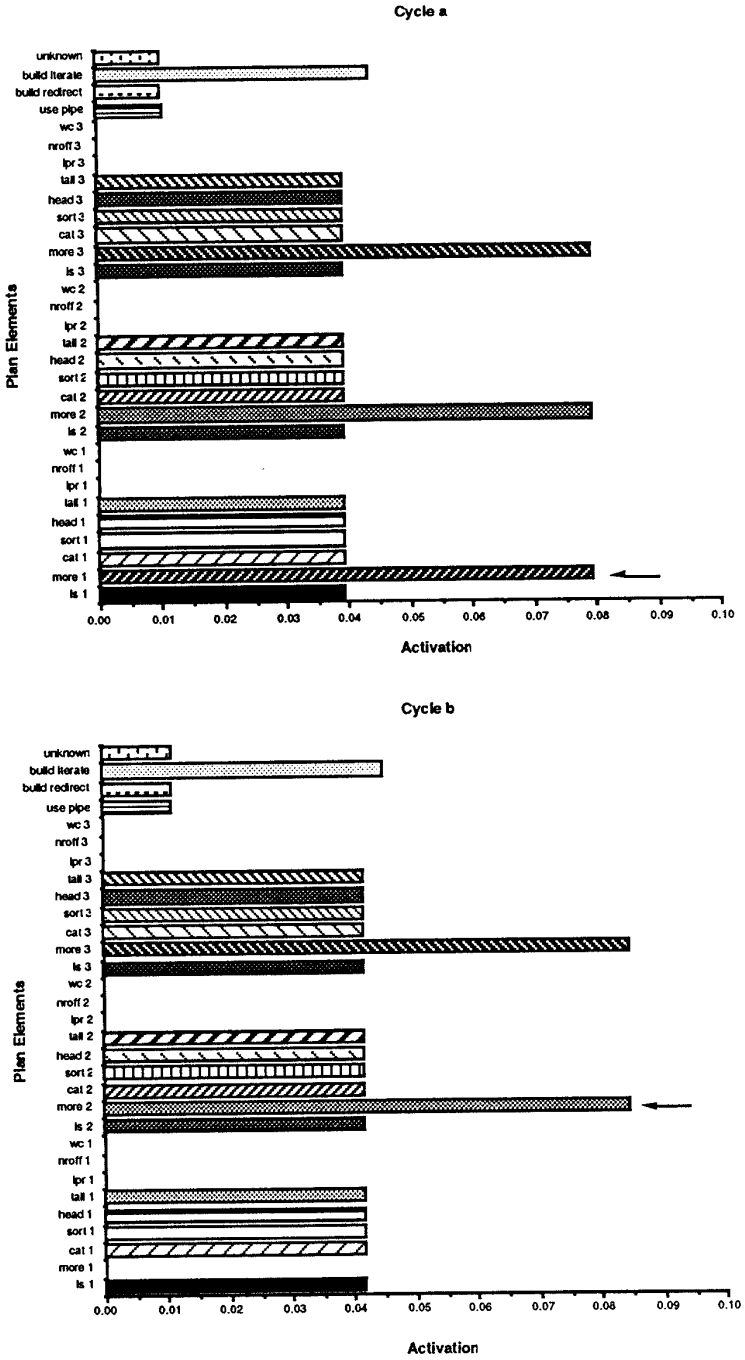
Iteratives. The iterative problem discussed here was described in English to subjects as "Display the contents of file OLD1, OLD2, and OLD3 one screen at a time". In this problem, three complete sets of single plan elements were created, with one set bound to each of the file names listed in the request.

In the case of the expert, the system has the knowledge to successfully produce the iterative command. A trace of the problem solution is shown in Figures 8a-8d. Initially, the most activated plan elements are the **more** plan elements that are bound to OLD1, OLD2, and OLD3 (the three plan elements are equally activated). However, none of the single plan elements will by themselves fulfill the requested outcome, and this distinguishes the iteratives from the multiple problems. For the iteratives, an action-planning phase is needed, since no one plan element whose preconditions are met can execute. At this point, it is arbitrary which of the plans are chosen for execution first, and UNICOM chooses to execute the plans in alphabetical order (Figure 8a). The first plan element selected for execution in the action-planning phase is the **more** plan element which is bound to OLD1. Its preconditions are met, it is executed, and its outcome (EXIST DISPLAY CONTENTS OLD1 ONE-SCREEN-AT-A-TIME) is added to the INTHEWORLD knowledge. A trace of this action is formed. It is initially bound to all three composite plans. The next integration step selects the **more** plan element bound to OLD2 for execution, its outcomes are added to the INTHEWORLD facts and to the trace. On the third integration phase, the **more** plan bound to OLD3 is executed, its outcomes are added to the INTHEWORLD facts and the iteration trace is updated.

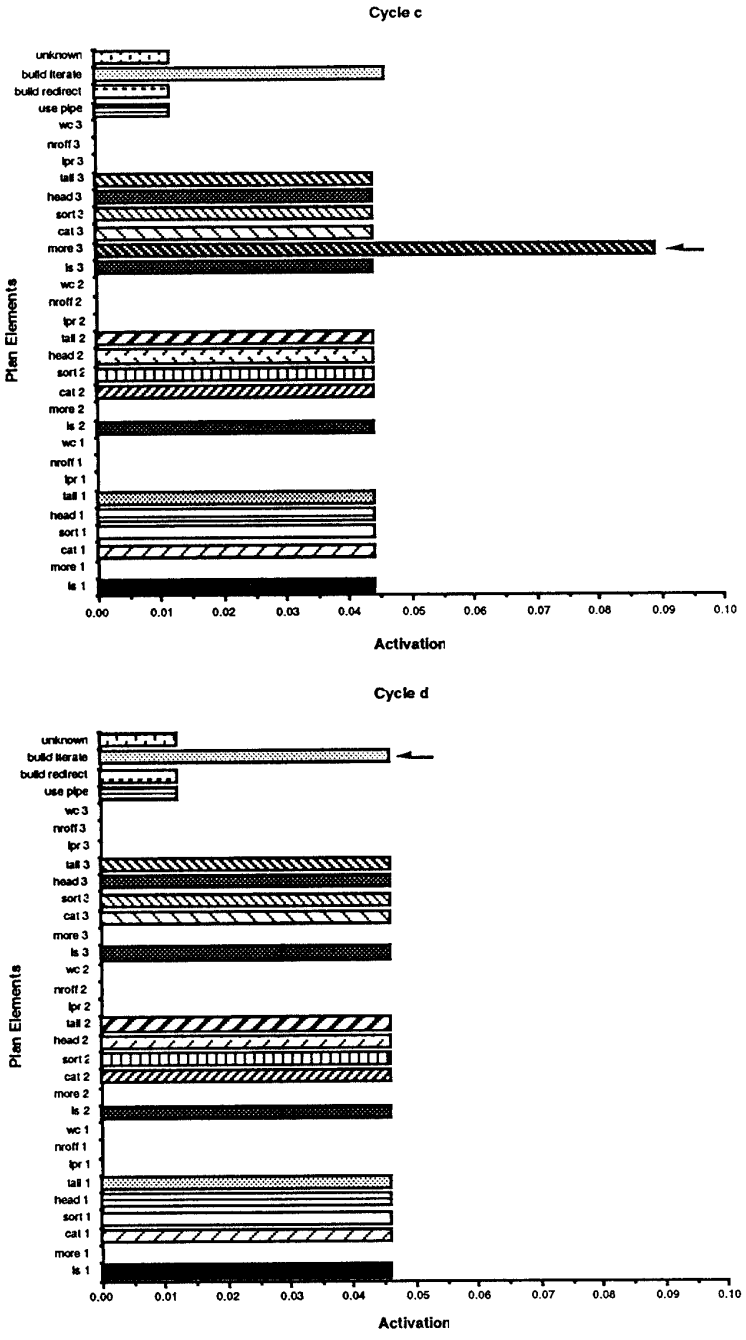
At this point, a plan that builds an iterative command is selected for execution, in a manner analogous to the BUILD-PIPE plan in (7), generating PLAN[BUILD-ITERATE **more** ^OLD1 **more**^OLD2 **more**^OLD3], which becomes the dominant plan on the last integration cycle (Figure 8d), and **more** is executed on OLD?, completing the task. Again, we are modeling the expert user in this case, and that knowledge base includes knowledge of the ? symbol for iterating commands.

Simulating the Less Expert User

The knowledge base for the simulation of the prototypical expert includes all facts required to successfully produce all of the commands simulated. To simulate the



Figures 8a and 8b. Activation values of plan elements and plans for cycles a and b from the UNICOM simulation of the **more** iteration task.



Figures 8c and 8d. Activation values of plan elements and plans for cycles c and d from the UNICOM simulation of the **more** iteration task.

performance of intermediate and novice users we make the assumption that such users lack knowledge. Three kinds of deficiencies are considered: users may not know all the plan elements; their knowledge about the various redirection commands may be incomplete; these deficiencies are simulated by appropriately lesioning UNICOM's knowledge base. The third deficiency may reflect either a lack of knowledge or a short-term memory problem: users fail to build sufficient action traces during planning.

As previously stated, in our simulations of intermediate and novice users we have not added misconceptions to the data base, but only deleted various facts or strategies from the perfect expert repertoire. Surely users have misconceptions, but our simulations merely explore the consequences of missing knowledge.

Single and multiple commands. UNICOM can perform single and multiple commands when it is stripped of general and specific knowledge about redirecting input and output of commands. All it needs to know is the specific commands themselves. Failure occurs when some command is unknown.

Composite commands. Users with an intermediate level of experience in Doane et al. (in press) typically knew about redirection in general, and knew specifically how to redirect input and output between commands and files, but did not know how to use the pipe to redirect input and output between commands. In the simulation enough knowledge was retained to select the BUILD-REDIRECT plan for execution, but not the BUILD-PIPE plan. We describe how UNICOM performed a composite task with this lesioned knowledge base.

The composite task description was identical to the one discussed above in the section on composites, that is, to "Display the first-ten alphabetically arranged file-names in the current directory". Indeed, the only change made from the simulation of expert performance was that we removed the plan for building pipes (7) from the knowledge base of the simulation as well as some facts from the UNIX fact base, such as the general facts that redirection of input and output can occur between commands through the use of pipes, and any pipe redirection facts specific to commands (e.g., that **ls** output can be redirected to another command). Other facts about redirection were retained. Thus, UNICOM still knew about redirecting input and output in general, it retained a BUILD-REDIRECT plan using redirection symbols (e.g., >, <), as well as specific facts for each command relating to redirecting its output to a file (e.g., the fact that **ls** output can be redirected to a file).

This intermediate model initially performs in a similar way as the expert model:

(a) In the first cycle **ls** is selected for execution, with the result that its outcome [DISPLAY FILE^USER] is added to the task net as an INTHEWORLD proposition, and the trace TRACE[COMMAND1^ls] is formed.

(b) Next, **sort** is selected for execution, producing INTHEWORLD[FILE^USER-ALPHABETICALLY-ARRANGED], and binding **sort** as the second command to the action trace.

(c) Finally, **head** produces the requested outcome, and is added to the trace.

(d) The conditions for BUILD-REDIRECT are now satisfied (all of the required knowledge about REDIRECT is indeed part of the task net), UNICOM is done with planning, and BUILD-REDIRECT generates new plan elements to redirect.

(e) BUILD-REDIRECT creates three composite plan elements that together will complete the requested composite sequence using redirection to files. Specifically, the plan elements created produce **ls**>TEMP1, **sort** TEMP1>TEMP2, **head** TEMP2.

Steps (a) to (c) are the same as in the expert simulation. The difference is only in (d) where BUILD-REDIRECT was generated instead of BUILD-PIPE. Using redirection in this way was often observed in intermediate users in Doane et al. An expert user would not choose to use redirect because they were asked to use the least number of keystrokes as possible to complete their productions, and this is accomplished through use of the pipes. If the user has knowledge of pipes (e.g., an expert), then they should use pipes to complete their production. A simulation of novice performance can be obtained by removing the knowledge about redirection altogether. In this case, the system fails, and no answer is produced.

What these simulations show is that the psychological processes involved in building redirection plans for composites are still quite complex. The creation of a redirected composite command requires a considerable amount of knowledge. Furthermore, the planning process for construction of redirected composites makes heavy demands on working memory, due to the need to track the trace of commands, similar to that for building piped composites.

Iteratives. As a second illustration of how UNICOM performs when certain pieces of knowledge are missing from its long-term memory, the iterative problem that was discussed above in the expert iterative section will be reconsidered. The task is to display the contents of files OLD1 OLD2, and OLD3 on the screen one screen at a time. The simulation is the same as before except for the formation of the action trace. We have assumed that intermediate users often do not build sufficiently precise action traces, e.g. they remember only that **more** had been performed, instead of the specific **more**^OLD1. However, as soon as OUTCOME[**more**] becomes part of the network it will inhibit all plan elements with **more** as their outcome, regardless of the file bound to the **more** plan element. As a consequence, the iteration cannot be performed. (The more specific OUTCOME[**more**^OLD1] will only inhibit the **more** operation on OLD1, on the other hand). Thus, the system fails because of sloppy record keeping.

If outcomes are tracked correctly, on the other hand, and the trace TRACE[**more**^OLD1 **more**^OLD2 **more**^OLD3] has been formed, intermediate users may still differ from experts in that they do not have the knowledge necessary to use an iteration symbol. If this knowledge is present, an iteration-plan will be constructed, resulting in the command **more**^OLD?. If not, three separate **more** operations will be performed on the three OLD-files. Either way constitutes a correct solution to the problem, but the former is simpler and more elegant and therefore preferred by experts.

Thus, intermediate and novice performance, according to these simulations, is deficient for three reasons: some plan elements may be missing; knowledge about redirection may be missing to various degrees; and imprecise memory traces may be formed.

Comparison of the Model with the Group Performance Data

On single or multiple problems UNICOM fails only when it does not know the commands involved. That is precisely what we observe in the Doane et al. (in press) data.

We can predict performance on multiples if we know how subjects do on singles (Figure 2).

Doing the same thing three times is not as easy as doing it only once for iterative problems (Figure 1). An analysis of what UNICOM must do to perform correctly on iterative commands shows why this might be so. In order to perform the same command three times, one needs to know not only the command itself but must also possess a strategy for accurate record keeping: if, after printing a file, the system merely remembers that a file has been printed, without specifying which particular file, this outcome will inhibit printing files in general, not just the particular file that has just been printed. This probably reflects a short-term memory problem: inexperienced users who might encode outcomes separately for the command used and the object produced, are likely to strain their short-term memory resources. Our model strongly suggests that becoming an expert UNIX user involves acquiring appropriate strategies for chunking in short-term memory, in addition to everything else. Note that keeping track of intermediate results is a problem not just for novice subjects, but also for more experienced subjects. As previously stated, even those users who know how to use iteration symbols (instead of merely repeating the same command three times) find iteration problems difficult, though they do better than users who do not know how to use iteration symbols at all.

Composite problems are by far the hardest for the model, and even the experts studied in Doane et al. are far from perfect. The analyses presented here would lead one to expect such a result, for the demands, both in terms of knowledge about UNIX and processing strategies and resources are considerable ones. The knowledge prerequisites for building pipes or redirection are high, for it is not just general knowledge that is required, but a lot more specialized information about what can be redirected where and how. This is surely a major reason why composite problems are so difficult, and why knowing the component commands themselves is only one, far from dominant, prerequisite for success (Figure 1).

On the other hand, the problems users have with iteratives, which we propose are due to faulty processing strategies and (or) short-term memory resources, indicate that lack of knowledge may be merely one reason for the difficulty of composite problems. Accurate tracking of intermediate results is at least as crucial here as with iterative problems. In order to use pipes and redirection, it is not enough to remember that the commands **ls** and **head** were used, but the model suggests that one must know precisely what the output of one was, and the input of the other. Therefore, an expert user must acquire suitable short-term memory encoding strategies as an important component of the skill of using UNIX. In the Doane et al. data instances were observed where the commands in a composite task were given out of order. These might be due to cases where subjects' memory traces were insufficiently precise, just as we have hypothesized for iterative problems. This idea is consistent with other research findings. It has also been observed in other studies (e.g. Anderson & Jeffries, 1985) that novice performance may suffer from loss of information from working memory.

UNIX knowledge and UNIX strategies are acquired in parallel, of course. Just as with iteratives, more experienced users who know about pipes make fewer errors in using them than intermediate users who rely on the less advanced redirection method.

Before executing a composite plan, UNICOM engages in a great deal of planning. It is interesting to note, therefore, that expert subjects took significantly longer to begin composing a composite command (i.e., to make an initial keystroke) than to begin composing a single or multiple command (Doane et. al., in press). Their mean reaction time

was 11.25 seconds for composites versus 5.58 seconds for singles. This pattern was not quite as pronounced with novices, which suggests that they did not complete an extended planning phase. Mean reaction times were 8.30 seconds for composites and 5.14 seconds for singles for novice subjects. This interaction was statistically significant and in particular, experts took significantly longer to start composing than novices.

So far, UNICOM has matched the group expert-to-novice data reasonably well, based on informal analyses. We now turn to the simulation of the acquisition of UNIX knowledge in individual users. To account for the group data, we have made assumptions about what experts, intermediates, and novices know. That is, we have deleted knowledge according to our intuitions and according to averaged group data. We now turn to simulations where we are not assuming particular knowledge: We actually have data on what an individual subject knows. These individual simulations serve to confirm the assumptions made on the basis of the group data.

Long-term Knowledge Acquisition in Individual Users

We simulated the individual performance of four of the users that Doane et al. observed at two different points in time, two years apart. One of the subjects was a novice when first observed, and was still classified as a novice two years later (N-N), the second subject was a novice at first but was then classified as an intermediate (N-I). The third subject was classified as an intermediate initially, and then as an expert (I-E), and the fourth subject was classified as an expert both times (E-E).

For each of these users we built a long-term memory network that represented his or her knowledge about UNIX, both for the first and the second test two years later. This was done by examining the commands that the user could actually perform and adding to that user's knowledge base all those facts that our task analysis identified as necessary for performing these commands. For example, if a user created the command **ls|sort**, then his or her data base would include the fact that he or she knew both **ls** and **sort**, and the fact that he or she knew about redirection and pipes in general, that he or she knew the specific redirection fact that the output of **ls** could be redirected to a command, as well as the fact that **sort** could receive input from a command.

Users had to explicitly show that they knew how to use iteration, redirection, and pipes before we included the corresponding BUILD-ITERATION, BUILD-REDIRECTION, and BUILD-PIPE plans in their knowledge base. That is, if a user showed the ability to use pipes, then their UNICOM model included a BUILD-PIPE plan. For all of the users modeled below, we included this plan, since they all showed evidence of pipe use in their productions. Thus we are not modeling failures to execute composite commands based on the lack of a BUILD-PIPE plan.

We are, however, modeling errors that take place because of lack of knowledge about specific commands that might be bound to the BUILD-PIPE plan. For example, a user might know the command **ls**, but not know that its output can be redirected. If this is the case, their BUILD-PIPE plan will never be selected for execution in a task such as **ls|sort|head**, since its prerequisites would never be satisfied.

Below we describe our investigation of what UNICOM would do, given the knowledge base appropriate for each modeled user, on our standard composite command, the **ls|sort|head** task. We performed these simulations to compare the simulation's performance with what the subjects actually did.

Subject N-N. He knew **ls**, but did not know **sort** or **head**. Although he knew about redirection in general, and knew how to redirect output to a file, he only displayed this knowledge in the context of the **cat** command.

With this impoverished knowledge base UNICOM could not solve the **ls|sort|head** task. The activation values for the plan elements after the first integration cycle appear quite random. The command **ls** is no more activated than some of the other irrelevant commands. This is the case because the **ls** plan element is not receiving any activation through causal chaining from the (nonexistent) **sort** and **head** plans elements. The system fails, and no answer is generated.

Two years later, N-N still displays minimal knowledge. He is still a computer science major, and still using the system, but is still unable to perform the **ls|sort|head** task. Indeed, in terms of the knowledge relevant to this task, N-N's knowledge base has not changed, and UNICOM fails in the same way as before.

Subject N-I. On the first test, this novice user had no knowledge of **sort** or **head**, but he did have knowledge of redirection in general, and showed evidence of knowledge about pipes and redirection to files. The system fails to produce an answer as in the previous case.

On the second test, N-I, now an intermediate, entered **ls** as his response for this task. The user now has knowledge of **sort**, and this addition to its knowledge base allows UNICOM also to generate **ls**, rather than UNKNOWN, as before. Although the command **ls** was available before, it was not preferentially activated. Now, however, it becomes more strongly activated because the **sort** command sends it activation since the outcome of **ls** is an unmet precondition of **sort**. Thus, both for N-I and UNICOM, knowing only **ls** results in the generation of a null response, while knowing both **ls** and **sort** results in the response **ls**.

Subject I-E. On the first test, I-E was classified as an intermediate user, but did not have knowledge of **sort** or **head**. I-E did have knowledge of pipes and redirection. Both I-E and UNICOM generated the UNKNOWN response in this situation.

On the second test, I-E had gained the knowledge to produce the command **ls|sort|head**. Hence UNICOM was given all the knowledge relevant to this task. Thus, UNICOM performs as in the case of the ideal expert. Basically, the system traces through all of the planning stages, and then builds a pipe for the command it has chosen.

Subject E-E. Subject E-E, an expert user, entered **ls|tail** for this task on the first test, an incorrect answer. The knowledge base for this user includes the commands of **ls**, **sort**, and **head**, knowledge about redirecting input and output in general, and knowledge about redirecting **ls** output to a command. Beyond that, his redirection knowledge is limited to redirecting **sort** input from a file, and redirecting **head** input from a file.

With this knowledge base, UNICOM produced the **ls** command on the first cycle, but does not explain why the user entered **tail** after **ls**. Intuitively it is, of course, obvious that **head** and **tail** are related, even though they perform opposite functions (**tail** lists the last ten lines in a file, rather than the first ten lines, as requested), but this fact is not represented in the model.

On the second test, E-E demonstrated that he knew how to produce **ls|sort|head**. Thus, UNICOM is given all the knowledge it needs and performs like the ideal expert.

Adequacy of the simulations of individual data. If UNICOM is given the same knowledge as individual subjects appear to have at different stages of skill, UNICOM performs the same way as the subjects do, up to a point. It has no error theory in the sense of Buggy (Brown & Van Lehn, 1980) because we have not given it any misconceptions, either about individual plan elements or redirection facts in general. We have merely taken away knowledge to simulate intermediate and novice performance. Therefore, we can only account for those errors that subjects make based on lack of information. We cannot account for errors that subjects make based on misconceptions. Apart from this limitation, however, the model simulates performance data rather well, both for the group data and for individual users.

Below we discuss the implications these results have for the design of the UNIX system. Following this discussion, we summarize our conclusions.

Design Implications

The UNIX operating system is a very powerful tool - but not an easy one to use. Norman (1981) pointed out that the standard UNIX interface of the early eighties had terse commands, lacked feedback, and presented almost insurmountable barriers to novices attempting to use the system. But although he was very critical of many of the basic features of the UNIX user interface, he had high praise for many of the basic architectural features of the UNIX operating system. Norman pointed out that UNIX provides a user with the capability to combine basic commands using input and (or) output redirection into composite commands to carry out more complex tasks.

Today, a number of new graphical interfaces are being developed to facilitate the use of UNIX. However, these do not directly aid composing complex commands. Our analyses suggest that it is precisely in that area where users need help. Specifically, even experienced users of the UNIX user interface have trouble making effective use of the building block philosophy that is a core feature of the UNIX operating system. Our theoretical analyses suggest that there are several potential causes that prevent novices and intermediates from making effective use of this tool. The first is working memory load, the second problem involves conceptual difficulties, and the third anticipating the outcome.

Memory Load

The process used by the model to generate composite commands involves two phases. The first phase is comprised of identifying the correct commands, sequencing them properly, and retrieving facts that are relevant to the execution of the build pipe plan. It may be the case that intermediates have all of the knowledge necessary to successfully execute a plan to build a piped command, but they simply can't assemble all of this knowledge in working memory and properly sequence the commands in order to successfully generate a composite. Experts are able to generate composites because all of this knowledge is highly overlearned and they are skilled at properly sequencing commands. This argument would suggest a set of essential augmentations to a graphical interface that would provide the memory aides that might enable some levels of users to successfully keep track of intermediate results when generating composites.

Conceptual Difficulties

The second set of problems appears to be primarily conceptual. Note that the build pipe plan has as prerequisites that users have explicit knowledge of the redirection properties of the individual commands that go into a composite. Furthermore, the notion of pipe and input and output redirection in general is highly abstract. Many novice and intermediate users simply do not have the correct detailed conceptualization of input and output redirection nor do they have specific detailed knowledge especially of individual commands that is required to successfully employ pipes in creating composites. The family of possible design solutions suggested by these conceptual difficulties would consist of attempts to develop extensions to a user interface that present the notion of redirection in some concrete graphical representation.

Anticipating the Outcome

Consider how the expert simulation solved the composite problem. A basic element for the whole process was the precise anticipation of the outcome of the task the system was instructed to do. It is not merely that UNICOM must know the outcome so it knows when to stop. Inferring that a specific file DIRECTORY-ALPHABETICALLY-ARRANGED was part of the outcome played a crucial role, because it made UNICOM generate a set of plan elements bound to that file, some of which were highly activated by the instructions, and which then in turn activated other plan elements which they needed as preconditions. That is why the system started by listing the contents of the directory, something not called for in the instructions but a necessary step in the solution of the problem. An inexperienced user who does not anticipate the desired outcome with enough precision has, according to our model, no chance for a correct solution.

Discussion

Understanding an instruction means being able to produce an appropriate plan for action. The construction-integration model of comprehension provides one account of how this understanding can be achieved computationally. What we have shown here is precisely what knowledge users of UNIX must have in order to produce commands, and what steps are involved in command production, according to this theory. Of particular interest was the question how the powerful complex commands that UNIX affords are composed from simple basic level commands. We have seen that a great deal of specific and general knowledge was necessary for that purpose, and we have analyzed in detail just what the resource requirements are for producing these commands.

Our model was based on a cognitive task analysis which we embedded within the framework of a general theory of comprehension -- the construction-integration theory. At one level, what our model achieves is a demonstration that the knowledge and information processing steps that we have described are indeed sufficient computationally to produce the UNIX commands specified by the instructions. We can make stronger claims than that, however, because we were able to tie our model simulations to experimental data collected from UNIX users.

The main features of the experimental data concerned, on the one hand, performance on different types of problems, and on the other, performance by users at different levels of expertise. The most salient observation concerning problem types is that single and multiple commands make quite different demands on users than do composites and iteratives. The model presented here gives a good account of these differences: it

specifies precisely the locus of these differences both in terms of knowledge prerequisites and processing strategies. It also provides an explanation for the differences that have been observed in the ability of novice, intermediate and expert subjects to produce UNIX commands. Many salient differences between these user populations can be simulated by deleting certain knowledge requirements and processing strategies from the user's repertoire.

In terms of the continuing discussion about the nature of expertise, the present results come down heavily on the side that explains the development of expertise as a monotonic accrual of factual knowledge and processing strategies. Intermediates and novices fail in certain situations because they do not have all the pieces of knowledge that are necessary that experts can rely upon (e.g., they don't know exactly what inputs are allowed for **head** in a pipe), and they employ deficient strategies (e.g., action traces are encoded in memory in such a way that they will become confused). Thus, in spite of the undeniable fact that expert and novice performance look quite different, the present model accounts for these differences without recourse to qualitative process distinctions, or global knowledge restructuring. Qualitative differences, in our model, result from a monotonic accrual of facts.

The model presented here is still incomplete as a simulation of how users produce UNIX commands. It does not provide a complete account for errors (e.g., it does not tell us why **tail** was used where **head** should have been used). Further error analyses would undoubtedly prove extremely informative psychologically, as they have in other domains (e.g., Cummins, Kintsch, Reusser, & Weimer, 1988). However, for some purposes even the present incomplete model has important implications. We have discussed above some design implications of our results. There exists today an enormous amount of interest in the design of graphical user interfaces for UNIX, and a great deal of activity. We do not claim that the recent developments in this field are not worthwhile; quite on the contrary, they address many of the problems with UNIX that Norman (1981) identified. But significantly, they are not at all concerned with the issues we have addressed here. A better theoretical understanding of what UNIX users actually have to do could have a considerable impact on the way design questions are approached today. Similarly, we would claim that our results have non-trivial instructional implications. The enormous popularity of the UNIX operating system raises urgent questions about the effectiveness of instruction, and once again, the present results may help to redirect these concerns in significant ways. An incomplete processing model such as the one presented here may be quite sufficient for this purpose, as has been argued in another context by Nathan, Kintsch, and Young (1989).

The model proposed here should be regarded as a hypothesis about command production. So far, the empirical tests of that hypothesis have been encouraging, but claims could easily be exaggerated in this respect. We have already noted the incompleteness of the model, as a result of which more detailed empirical evaluations are made difficult. In addition, the best way to test a model's adequacy is to contrast it with alternative models. There is no question that alternative models for the command production task could be developed from other conceptions of problem solving. Our interest here was a more modest one: we wanted to explore the feasibility of one account provided by the construction-integration theory.

Problem solving has typically been regarded as the prototype for cognition. Some areas of cognition, however, may be better studied as problems in comprehension and understanding, more akin to perception than problem solving in the tradition of Newell and Simon (1972). The construction-integration theory, originally developed for discourse

understanding, provides a framework to explore the explanatory power of the notion that understanding may be another pivotal concept in the study of cognition. It does this by combining the advantages of symbolic theories of cognition (representational mechanisms, planning systems) with the advantages of connectionist techniques (constraint satisfaction, parallel processing). Holyoak (1989) has discussed the promise that such approaches which he has called "symbolic connectionism" hold. The ability of the model presented here to provide a principled (in terms of the construction-integration theory) account for the quite non-trivial task of how people produce UNIX commands in response to verbal instructions demonstrates once more the power of this symbolic connectionism.

References

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. Memory and Cognition, 9, 422-433.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. Journal of Experimental Psychology: Learning, Memory, and Cognition, 10, 483-495.

Anderson, J. A., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. Human-Computer Interaction, 1, 133-161.

Brown, J. S., & Van Lehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. Cognitive Science, 4, 321-398.

Card, S. K., Moran, T. P., & Newell, A. (1983). The psychology of human-computer interaction. Hillsdale, NJ: Erlbaum.

Chi, M. T. H., Feltovich, P. J. & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. Cognitive Science, 5, 121-152.

Cummins, D. D., Kintsch, W., Ruesser, K., & Weimer, R. (1988). The role of understanding in solving word problems. Cognitive Psychology, 20, 405-438.

Doane, S. M. (1986). A longitudinal study of UNIX user's expertise, UNIX mental models, and task performance. Unpublished doctoral dissertation, University of California, Santa Barbara.

Doane, S. M., Pellegrino, J. W. and Klatzky, R. L. (1989). Mental Models of UNIX as Revealed by Sorting and Graphing Tasks. Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, Kailua, Kona, Hawaii: IEEE Computer Society.

Doane, S. M., Pellegrino, J. W., & Klatzky, R. L. (in press) Expertise in a computer operating system: Conceptualization and performance. Human-Computer Interaction.

Holyoak, K. J. (1989). Symbolic connectionism: A paradigm for third-generation theories of expertise. To appear in K. A. Ericsson & J. Smith (Eds.), The study of expertise: Prospects and limits. MIT Press.

Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. Paper presented at the AERA annual meeting, March, 1982.

Kieras, D. E., and Polson, P. G. (1985). An approach to the formal analysis of user complexity. International Journal of Man-Machine Studies, 22, 365-394.

Kintsch, W. (1988). The use of knowledge in discourse processing: A construction-integration model. Psychological Review, 95, 163-182.

Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. Psychological Review, 85, 363-394.

Koedinger, K. R., & Anderson, J. R. (1989). Perceptual chunks in geometry problem solving: A challenge to theories of skill acquisition. Proceedings of the 11th Annual Meeting of the Cognitive Science Society. Ann Arbor, Michigan: Erlbaum

Larkin, J. H. (1983). The role of problem representation in physics. In D. Gentner and A. Stevens (Eds.), Mental Models (pp. 75-98). Hillsdale, NJ: Lawrence Erlbaum Associates.

Larkin, J. H., McDermott, J., Simon, D., & Simon, H. A. (1980). Expert and novice performance in solving physics problems. Science, 208, 1335-1342.

Kintsch, W., Welsch, D., Schmalhofer, F., & Zimny, S. (in press). Sentence memory: A theoretical analysis. Journal of Memory and Language.

Mannes, S. M. & Kintsch, W. (1989). Planning routine computing tasks: Understanding what to do. Manuscript submitted for publication.

McKeithen, K. B., Reitman, J. S., Rueter, H. H. & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. Cognitive Psychology, 13, 307-325.

Nathan, M. J., Kintsch, W., & Young, E. (1989). Towards a theory of problem comprehension and its implications for an unintelligent tutoring system for word algebra problem solving. Manuscript submitted for publication.

Newell, A., & Simon, H. A. (1972). Human problem solving. Englewood Cliffs: Prentice-Hall.

Norman, D. A. (1981, November). The trouble with UNIX. Datamation, 139-150.

Rosch, E. (1975). Cognitive representations of semantic categories. Journal of Experimental Psychology: General, 104, 192-223.

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, (1986) Parallel distributed processing: Explorations in the microstructure of cognition (Vols. 1-2). Cambridge, MA: MIT Press.

Schank,R. C. (1982). Dynamic memory: A theory of reminding and learning in computers and people. New York: Cambridge University Press.

Simon, H. A. (1986). The information processing explanation of Gestalt phenomena. Computers in Human Behavior, 2, 241-255.

van Dijk, T. A. & Kintsch, W. (1983). Strategies of discourse comprehension. New York: Academic Press.

Wertheimer, M. (1982/1945). Productive thinking. Chicago, IL: University of Chicago Press.