An Integrated Model of the Initial Acquisition and Retention of a Cognitive Skill

A Doctoral Thesis
by

George Essex Engelbeck

Department of Psychology
University of Colorado
1989

Institute of Cognitive Science
Technical Report #89-2

ABSTRACT

A model of the initial acquisition and retention of a cognitive skill was developed. This model is presented in the form of a computer simulation that learns by analyzing and storing the results of problem-solving episodes in an associative network. It incorporates an associative interference theory of forgetting and predicts that procedures will be difficult to retrieve when needed cues are unavailable or no longer make relevant discriminations.

Two experiments were used to test the learning and retention assumptions of the model. In these experiments participants performed tasks using a menu-based computer system. The first experiment used the data from Polson, Muncher, and Engelbeck (1986) to test the model's transfer assumptions and permitted a direct comparison of the model's performance with that of Kieras and Polson's (1985) Cognitive Complexity Theory. Both models provided excellent fits to the data.

The second experiment investigated the effects of cueing variant-inconsistent methods on transfer and retention. The empirical results showed: 1) Inconsistencies can lead to inappropriate transfer; 2) Explicit cueing of variant methods can reduce errors during transfer and retention; and 3) The retention errors observed on variant task steps were consistent with a classical interference account of forgetting.

The data from the second experiment were used to test the model's account for the initial acquisition, transfer, and retention of procedures. The model gave good fits to the observed task completion times and errors for both a training and retraining session. Deviations of the model's predictions from the observations, however, suggested that the encoding assumptions of the model overemphasized the role played by system states and that a greater encoding of task goals was needed.

Finally, it is concluded that the model provides needed guidance for designing machines and computer applications that are intended to be used with little or no training. These applications form a large and growing segment of machines used by people everyday.

ACKNOWLEDGMENTS

This dissertation has benefited from the encouragement and assistance of many friends, colleagues, and institutions. First and foremost, the efforts of my committee chairs need to be acknowledged. Gary Bradshaw devoted large amounts of his time reading, editing, and trying to improve my writing skills. His efforts have improved this dissertation and, hopefully, my writing skills. Peter Polson has been my advisor since my entry into graduate school: I have always benefited and learned from my interactions with Peter.

I am also indebted to the other members of my committee. I thank Clayton Lewis who from time to time acted as a third committee chair. His management skills and common-sense advice helped me complete the dissertation within the limits of a demanding schedule. I would also like to thank Walter Kintsch and Gary McClelland for their comments on this work and for serving on my committee.

Friends and colleagues at the University of Colorado and at U S WEST Advanced Technologies also deserve thanks: Catherine Marshall for providing a strong motive for bringing this work to closure; Terry Roberts for her support during some of the rougher times; and Scott Wolff, Jim Alexander, Mark Schlager, Peter Foltz, Rob Crutcher, Susan Davies, Emilie Young, and Liz Muncher for listening to long harangues and ventings of frustration.

Finally, I would like to thank the members of my family for their support. I thank my wife Laurie for support given throughout this long project, for sacrificed weekends, and for proof reading. I thank my mother, Anne Engelbeck, for her support during my long university career and for several loans given in times of financial need.

Parts of this work were supported by an International Business Machines Corporation grant to Peter Polson and David Kieras and National Science Foundation grant number NSF IB1-8722792 to Gerhart Fisher, Walther Kintsch, Clayton Lewis, and Peter Polson. The opinions and conclusions contained within this research are those of the author and not necessarily those of IBM or the National Science Foundation.

CONTENTS

CHAPTER I
INTRODUCTION

The present study is concerned with the acquisition and retention of procedural skill. Procedural skill is the ability to perform a task requiring a sequence of discrete actions smoothly and accurately. Examples of procedural skills include solving mathematics or physics problems, editing text using word processors, and writing computer programs. This study investigates the relationship between the acquisition and retention of these skills.

Cognitive psychologists have been studying procedural skills for many years. Numerous skill phenomena have been identified; including speed-up (Lewis, 1978; Newell and Rosenbloom, 1981); automaticity (Schneider and Shiffrin, 1977); and expert-novice differences (Larkin, McDermott, Simon, and Simon 1980; Chi, Fetovich, and Glaser, 1981). The course of skill acquisition has been characterized (Fitts, and Posner, 1967; Anderson, 1982; 1987a) and several process models have been developed to explain various skill phenomena (Anderson, 1982, 1983, 1987a; Card, Moran, and Newell, 1981; 1983; Laird, Rosenbloom, and Newell, 1986; Laird, Newell, and Rosenbloom, 1987).

However, researchers have neglected the retention of procedural skill. Indeed, many researchers act as if skills are never forgotten.[1] As we shall see, procedural skills *are* forgotten and, as a few studies have shown, forgetting can be systematic and rapid (Shields, Goldberg, and Dressel, 1979; Engelbeck, 1986).

The present study develops a cognitive model of skill acquisition and retention. This model is cast as a computer simulation that accounts for the forgetting of procedural knowledge. It concentrates on initial learning and forgetting of procedures. The model uses general problem-solving to complete its tasks and learns by storing the results of these problem-solving episodes in memory. It assumes that forgetting effects are the result of cognitive processing, predicting that forgetting effects are most likely to occur when procedural knowledge is poorly connected to other procedures, or when retrieval cues are unavailable or no longer make relevant discriminations.

This chapter begins with a discussion of why it is important to study skills. Next, follows a summary of the benefits that will result by studying the relationships between skill acquisition and skill retention. This leads to a brief survey of studies on skill retention. At the conclusion of this chapter is an overview of the remainder of the document.

Why It Is Important to Study Skill Acquisition and Retention

In a society as complex as ours, we rely on the ability of others to perform tasks we cannot perform. For instance, we are willing to be passengers on airplanes that we are unable to fly ourselves. We rely on the pilot and crew of the airplane to get us to our destinations safely.

[1]Retention and forgetting can be thought of as two sides of the same coin. The two terms will be used interchangeably.

Our reliance on others has now been extended to reliance on computers.

Computers pervade the daily lives of many people. Tasks are performed in the modern workplace with the assistance of computer applications. People use applications to edit documents, communicate with colleagues, track inventories, and keep accounts. Outside the workplace, automated services are common. Automated teller machines offer customers 24-hour banking. Phones services offer customers voice-mail, call forwarding, conference calling and many other services. National computer services, such as Minitel in France and PRESTEL in the United Kingdom, have been established. These services allow subscribers to read daily newspapers, check current stock quotes, and purchase airline, train, and theater tickets. Even the home is not immune to the spread of computerized automation. Now many household appliances contain microprocessors allowing ovens to be programmed to cook dinner, coffee makers to brew coffee, video recorders to record television programs, washing machines to wash, and lights to go on and off. The proliferation of these applications and services motivates people to acquire the skills needed to use them.

Computers are changing both the quality and the quantity of the procedural knowledge people possess. The demands of computerized environments means people must learn additional procedural skills. Yet the level of skill required is not great. In the past, people relied on experts to accomplish tasks, while today many of these same tasks have been automated. People must acquire the skill of *telling the computer* to perform these tasks.

Often the knowledge required for competent use of computer applications and services has proven difficult for people to learn and remember. Because of costs, historical accident, and technological limitations, these machines and services have complex, eclectic user-interfaces. Phone services provide an example of how difficult an interface to a service can be (see Norman, 1988). To put a caller on hold using my 12-button phone I must "depress the switchook for one second" and then press "*9". The "*9" is a command whose meaning is unclear since it appears in many other tasks and is never explained in the manual. I presume this command was preceded by a star "*" to differentiate commands from phone numbers. This procedure taken by itself may not be difficult to learn, but there are many others like it. Similar procedures initiate call forwarding, conference calling, automatic call back, and many others. Initiating these functions requires arbitrary and opaque procedures. I have never used them.

People have similar difficulties using computer applications. These difficulties have lead to the creation of the field of human-computer interaction. This field is a branch of applied cognitive science dedicated to discovering why computer applications are difficult for people to use and to improving the usability of these applications.

Computer applications and services have changed the quality of people's procedural knowledge. Because people are often infrequent users of computer applications and services, their proficiency using these applications and services is often low. We know from studying skill that practice is required to learn a skill. Deliberate practice, however, is impractical for many computer skills. Few people are willing to spend their time performing textbook exercises to become

virtuosos of automated teller machines, electronic mail systems, or video recorders. Furthermore, the use of these machines does not require expert levels of performance. For instance, the transfer of funds from a savings account to a checking account does not require the speedy, efficient execution of a sequence of actions. All that is required is a modest amount of competence: the ability to select correct actions and perform them at a reasonable rate.

Researchers have found that people are sensitive to the effort needed to master skills. Nielsen, Mack, Bergendorff and Grischkowsky (1986) found that most users ignore many functions offered by computer applications because they believe these functions would take too long to learn.

Skills cost time and money to acquire. When skills are forgotten the costs can be also be great: lives are lost when people trained to perform CPR forget the components of their skill. In addition, the forgetting of skills can result in the failure of a society to accept a new and valuable technology. The information services industry provides a good example of this problem. Information services, the business of providing large data-bases and data transmission capabilities to customers, was expected to become a billion-dollar-a-year industry by the late 1970s or early 1980s. The growth of this industry, however, fell far short of initial expectations. These services required a great deal of training to use, yet the procedures were quickly forgotten (Field and Harris, 1986). The difficulties people had learning and remembering how to use these services retarded the growth of this industry. Subscribing businesses lost time and money teaching their employees to use these services, and the benefits were never fully realized.

To summarize, there is an increasing pressure on people to acquire skills needed to use machines and services. The proliferation of these machines and services with their eclectic, stylized interfaces has changed the nature of skill acquisition. People are infrequent users of the skills needed to use these services. As a result, they never reach a great proficiency executing these skills.

The present study looks at the acquisition and retention of one skill. This study does not investigate a glamorous skill performed only by a handful of experts, nor a skill that takes years of dedication to acquire. Instead, a commonplace skill was selected: using a menu-based computer application. This skill is representative of skills people must acquire to perform everyday tasks using computers. It is also representative of the commonplace procedural skills people must learn in technologically complex societies.

The present study concentrates of the initial learning and forgetting of procedural knowledge needed to operate a menu system. Of particular interest is the initial encounter of users with the menu system. For many computer applications, users' initial encounters is the most important. If people cannot use many applications within a few minutes they are unlikely to continue or try again. Additionally, if they cannot remember how to perform tasks using an application, they can become frustrated and stop using the application.

Merits

A study of the relations between the initial acquisition and retention of skill has both practical and theoretical merit. On the practical side, the results of this study are important because procedural skills themselves are of great practical importance. What is learned about the acquisition and retention of procedural skills can be used to make these skills easier to acquire and retain. By focusing on the forgetting which results from cognitive processing, this study can help educators and designers identify and avoid forgetting effects. On the theoretical side, we can apply what has been learned about the retention of verbal materials to the retention of procedural knowledge. Forgetting effects allow us to impose a strong constraint on theories of procedural skill: Adequate theories must explain observed forgetting effects. The practical and theoretical merits of the present study will be considered in more detail below.

Practical Merits

Relationships between skill acquisition and skill retention have implications for the design of machines people use and for the instructions they are given. Designers cognizant of these relationships would be able to identify steps in procedures which are difficult to learn and remember. Systems could be restructured to avoid, or compensate for, these troublesome steps. Also, knowledge of the relationships between skill acquisition and skill retention would help educators make their instruction more effective. Educators would be able to identify and selectively train those components of skills that are most likely to be forgotten.

Theoretical Merits

This study makes two theoretical contributions to cognitive psychology. First, it generalizes what has been learned about the retention of complex verbal materials to the retention of procedural skills. Second, it shows that the forgetting of procedural skills can strongly constrain theories of skill.

Applying what is known about the retention of verbal materials to procedural skills. The few studies performed suggest that memory for procedural knowledge is similar to memory for complex verbal materials such as sentences and text (see Ericsson and Crutcher, 1988; Farr, 1987). This study assumes that the memory principles used to account for the forgetting of complex verbal materials can account for the forgetting of skills.

Hurlock and Montague (1983) have suggested procedural tasks are difficult to remember because they require the retrieval of a correct response and the discrimination of that response from a pool of possible responses. If this analysis is correct, then it should be possible to help people by making responses easier to remember and more distinctive. This requires a description of how people acquire, represent, and retrieve procedural knowledge.

Constraining theories of skill. Anderson (1987a, 1987b) has suggested that many different models of skill have been developed because investigators have only considered a small part of the available data. He argues that theories of skill should incorporate the constraint that the representations are constructible by a learning mechanism. This constraint broadens the amount of data a theory of skill would have to explain. Incorporating additional constraints in theory development and selection is a powerful principle. In keeping with this principle, the present study incorporates forgetting effects as an additional constraint on theories of skill.

Theories of Skill

Skill has been the focus of much research in cognitive psychology. Studies have explored the acquisition of skills such as geometry (Neves, and Anderson, 1981; Anderson, 1982), computer programming (Anderson, Farrell, and Sauers, 1984), calculus (Singley, 1987), physics (Larkin, 1981; 1983), radiological diagnosis (Lesgold, Feltovich, Glaser, and Wang, 1981), and text editing (Singley and Anderson, 1985; Kieras and Polson, 1985; Bovair, Kieras, and Polson, 1987). Studies investigated the differences between expert and novice representations of knowledge for a domain (Chase and Simon, 1973; Jeffries, Turner, Polson, and Atwood, 1981; Larkin, 1981; Chi, Fetovich, and Glaser,1981). Researchers have characterized the general course of skill acquisition (Fitts and Posner, 1967; Anderson, 1982, 1983, 1987a). General empirical phenomena such as speed-up (Crossman, 1959; Koler, 1979), automaticity (Shriffrin and Schneider, 1977; Anderson, 1982, 1987a) and the transfer of skill (Thorndike and Woodworth, 1901; Kieras and Bovair, 1985; Polson, 1988; Anderson, 1987a; Singley and Anderson, 1985; 1988) have been studied and characterized.

Skill has become a sophisticated area of study. Numerous models have been developed to explain skill phenomena. Notable models include: ACT* (Anderson 1982, 1983, 1987a); PUPS (Anderson 1987c; Anderson and Thompson 1987); SOAR (Laird, Rosenbloom, and Newell 1987); GOMS (Card, Moran, and Newell 1981, 1983), and Cognitive Complexity Theory (Kieras and Polson 1985; Polson 1988). While the number of models a field can produce may be evidence for its sophistication, it also indicates that there is a great deal of freedom in explaining phenomena.

The current state of computer models of skill resembles computer models of memory in the early 1970s. Then, Anderson and Bower (1973, page 465) complained that memory models "... have forgotten that people forget." Computer models of skill seem to have forgotten that people forget how to do things. They apparently assume that once procedural knowledge is acquired it is never forgotten.

Retention of Skill

This section reviews some of studies of skill retention. Of particular interest are studies that have shown that skills can be forgotten and that forgetting effects can be systematic (errors made during retraining are not random and occur in predictable locations).

In preface to this discussion, it must be noted that in contrast to the rich literature on skill

acquisition, only a few studies of skill retention have been performed (see Ericsson and Crutcher, 1988) and few attempts to model skill retention have been made. Several factors may underlie this puzzling gap. First, skill researchers have close ties to education. Educational institutions are, for the most part, evaluated on what their pupils acquire, not on what they retain. This emphasis may have carried over to the study of skill. Often, researchers investigating skill *acquisition* never repeat problems and hence do not distinguish incomplete initial learning from forgetting.

Another possible reason researchers have neglected skill retention is the field's preoccupation with expertise. Expert performance emphasizes the speed with which a person can execute a task. Studies of speed-up have had difficulty accounting for the initial acquisition of a skill (Lewis, 1978). Given the emphasis on speed of execution, the initial acquisition of a skill is frequently treated as a "warm-up period" and is omitted from any performance analysis. However, during period this "warm-up period" performance will be dominated by the retrieval and discrimination of correct responses.

Empirical measures related to forgetting such as recall rates have played a central role in the development of many psychological theories. For instance, models of text comprehension have relied on recall data as a major source of constraint. The text comprehension model of Kintsch and van Dijk (1978) was tested by comparing the model's predictions with recall of texts. This work assumed that difficulties in comprehending text would require additional processing or would result in deficient representations, affecting recall (Miller and Kintsch, 1981). Given the central role recall data has played in these areas, it is reasonable to believe similar data would be an important source of information about the acquisition, representation and retrieval of procedural knowledge.

Anderson (1987a, 1987b) has argued that current theories of skill are underconstrained by the available data. Skill retention data could serve to constrain theories of skill. Retention data have proven to be a valuable source of constraint in other domains. The role of retention data in the development of theories in other domains suggest that theories of skill will benefit by including skill retention.

Most studies of skill retention have assumed that accounts given for the retention of verbal material apply to skill retention (see Ericsson and Crutcher 1988; and Farr, 1987 for reviews). This assumption seems justified. When one looks at the retention curves for skills, there is little reason to believe skill memory is fundamentally different from memory for verbal materials. One striking parallel is the exponential shape of retention curves (see Ericsson and Crutcher, 1988).

Kolers (1979) provides a well-known description of the forgetting of skills. In his study Kolers had people practice reading inverted text. Initially, participants took a long time to read the inverted text. Participants took about 16 minutes to read the first page of inverted text compared to about 1.5 minutes for a page of normal text. Kolers had his participants practice reading the inverted texts until they were nearly as proficient reading this text as normal text (about 1.6 minutes per page). One year later, Kolers again gave participants practice reading the inverted texts. He found that they had retained a good deal, but not all, of their proficiency for reading inverted texts.

They took about three minutes to read the first page of inverted text. Kolers' study supports the conclusion that the components of a skill can be forgotten. Without data of this sort, a study of skill retention would be meaningless: there would simply be no phenomena to study. However, simple evidence of forgetting is insufficient to warrant investigation. Skills might be randomly forgotten, so that no analysis would be of interest.

Two studies have shown that the forgetting of procedures can be systematic. A study conducted by Shields, Goldberg, and Dressel (1979) represents a turning point in the study of skill retention. Shields and her colleagues tested the ability of soldiers to remember skills taught by the Army during basic training. Looking at the individual steps of tasks, they found that components of skill were not uniformly subject to forgetting. Certain steps were easily retrieved while others were commonly forgotten. An analysis of the forgotten steps revealed a systematic pattern: the steps most likely to be forgotten were "...steps not suggested by the previous sequence or by the equipment." Shields, Goldberg and Dressel (1979; page 4). They cited safety procedures as examples of procedures that were likely to forgotten. For instance, soldiers commonly forgot to check the backblast area of an artillery piece to see if it was clear. This step was not directly related to firing the artillery piece and was performed only as a safety precaution.

Engelbeck (1986) showed interference effects could be used to explain forgetting of steps needed to perform menu-based computer tasks. Participants in this study learned a sequence of tasks one day, then returned the next day to repeat the same tasks. Over half of the errors made by participants were made on four of 61 steps of the tasks. These four steps required participants to perform an action which received little use during the training session and occurred in a context where a more common method appeared to be appropriate.

A striking similarity exists between the pattern of forgetting in the experiments of Shields, Goldberg and Dressel (1979) and Engelbeck (1986) and the results found in studies of memory for verbal materials. These two studies show that memory for skill is influenced by the representations formed during acquisition and the cues available at retrieval. Shields and her colleagues found a lack of environmental cues was associated with forgetting effects. Cueing effects have been found in memory for lists and texts.

In a list learning study, Tulving and Pearlstone (1966) had participants memorize lists of words of varying length. Each word in the list was preceded by its category name. Participants were told that the category name did not have to be memorized. Two recall groups were formed. One group was given a sheet of paper and was asked to write as many of the words as they could remember on the sheet of paper. Another group was given a sheet of paper with the category names printed on it. These participants were also asked to write down all the words they could remember. Tulving and Pealstone found that the group given the category names recalled more words. In addition, when the first group was given another recall test using the category names, their performance improved over their first recall test. The category names clearly helped cue recall.

R. C. Anderson and Pitchart, (1978) demonstrated similar effects using short texts. In their experiment, participants read a story describing a house from either from the perspective of a burglar or the perspective of a potential buyer. After reading the texts, participants were asked to recall as much as possible. Anderson and Pitchart found that participants recalled information relevant to their perspective readily but omitted information relevant to the alternative perspective. However, when participants are asked to shift perspectives at recall they were able to recall additional information from the other perspective.

A second similarity between verbal materials and skills is the relationship between memory organization and recall. The method of loci serves as a clear illustration of organizational influences on recall. This mnemonic technique requires people to imagine themselves placing the items they wish to remember in a number of familiar locations. Having done this, they take an imaginary walk around these locations retrieving the items. The path structures their list recall.

Other memory organizations have been shown to be important for recall. The hierarchical organization of text propositions has been shown to be important predictor of memory for that text (Kintsch and Keenan, 1973). Similarity, Shields, Goldberg, and Dressel (1979) found that the goal structure was an important predictor of what procedures were likely to be forgotten. Recall that Shields and her colleagues found that safety procedures were often forgotten by soldiers performing tasks and concluded these procedures were forgotten because they were not directly related to the task being performed: Checking the backblast area was not directly related to the goal of firing an artillery piece.

Finally, Engelbeck (1986) was able to use a form of interference theory to explain the forgetting of procedures needed to perform menu tasks. Interference theory is one of two classical theories of forgetting and has been used to explain the forgetting of verbal materials (see Postman 1971; Postman and Underwood, 1973, for reviews) and more recently, to explain memory for sentences (Bower and Hilgard, 1981). Engelbeck's use of interference theory to explain the forgetting of procedures showed that the verbal learning literature was relevant to understanding the difficulties encountered by people acquiring procedures needed to operate computer applications.

In summary, the study of Kolers (1979) showed that skills are forgotten while the studies of Shields, Goldberg, and Dressel (1979) and Engelbeck (1986) showed that the forgetting of skills can be systematic and rapid. These results have important implications for the retention of everyday skills. They suggest that what psychology has learned over the past 100 years of research on human memory is relevant to understanding the retention of procedural skill and can contribute to the development of better models of skill .

Modeling the Retention of Skill

A procedure can be considered forgotten when it has been performed at one time but cannot be performed at some later time. It should be emphasized that this definition equates forgetting with failure to retrieve an item from memory. A retrieval failure does not necessarily mean that that

an item has been lost from memory; it might only be temporarily inaccessible.

Two explanations for forgetting were developed in the verbal learning literature. Trace-decay theory states that forgetting occurs when unspecified organic processes cause memories to degenerate to the point where they can no longer be retrieved. Interference theory states that forgetting occurs when memories similar to another memory prevent it from being retrieved. This is a cognitive explanation of forgetting and has served as the foundation for attempts to model procedural forgetting.

Attempts to model the retention of procedural knowledge have used explanations based on classical interference theory to explain why procedures are remembered or are forgotten. This section reviews Waern's (1985) suggestion that interference theory could be used to explain the forgetting of procedures and Engelbeck's (1986) use of Waern's suggestions to account for the forgetting of procedural knowledge necessary to perform several menu-based computer tasks.

Waern (1985) suggested that an interference explanation could be applied to the forgetting of procedural knowledge. In particular, she suggested that the terms used to describe transfer studies in the paired-associate learning literature could be mapped onto the production system formalism used to describe procedural knowledge. This mapping could predict forgetting effects. Before we see how this works, a short review of the these two formalisms is needed.

Paired-associate learning studies make a distinction between *stimuli* and *responses*. The pair A-B stands for a stimulus A paired with a response B. This notation describes the design of different transfer experiments. For example, consider an experiment in which participants learned two distinct lists: A-B, followed by C-D. The first pair, A-B, represents a list where participants learn to give the responses B to the stimuli A. The second list, C-D, represents a list that contains a new set of stimuli C and a new set of responses D. Because the lists are independent, little interference is expected to affect the C-D list learning. Table 1-1 lists basic transfer designs used in paired-associate learning (based on Kintsch, 1970, page 29), along with commonly observed results.

Table 1-1.
Basic Transfer Designs

List 1	List 2	Description	Result
A-B	C-D	Basic control list	No interference
A-B	A-C	Different responses to the same stimuli	Moderate interference
A-B	C-B	Same response to new stimuli	Low interference
A-B	A-Br	Stimuli and responses are re-paired	Hight interference

We now turn to a short review of production systems. Production systems are a common formalism used to model psychological processes (Newell and Simon, 1972; Anderson, 1976, 1983; Kieras and Polson, 1985; Laird, Rosenbloom, and Newell, 1987). A production system is composed of individual units of procedural knowledge called *production rules*. A production rule is a condition-action pair of the form: IF condition(s) THEN action(s). The conditions must be satisfied before the actions of the rule can be executed. The action component is executed when the conditions of the production are satisfied. Production rules can be viewed as modern psychology's stimulus-response pairs. Each production associates a set of conditions with a set of responses.

With this background, we are ready to show how production system notation can be mapped onto paired-associate terms. Consider the production IF (A) THEN (B). This rule can be cast as a paired-associate where the condition A is the stimulus and the action B is the response. This yields the paired-associate A-B. From this characterization interfering productions can easily be identified. Interfering productions are those which share a common goal and conditions (A) but have different actions (C). This corresponds to the AB AC condition in Table 1-1 where different responses must be given to the same stimulus.

A further assumption allows specific predictions of forgetting to be made. Waern (1985) noted that production rules can vary in their relative frequency of use across tasks. She suggests that when two or more production rules interfered with each other, the rule appearing with the greatest frequency would dominate and suppress the recall of less frequent rules. This is a form of McGeoch's (1932) response competition model of forgetting. McGeoch's model states that responses available at retrieval time compete with each other to be executed. On average, the expected winner of the competition is the response with the greatest associative strength.

Waern's characterization of production rules as paired-associates changes the meaning of the paired-associate notation in two ways. Originally, a paired-associate (e.g., A-B) was used to denote a *list* of items. Waern's characterization applies to *individual productions* within a production system. Also, the conditions of productions make statements about the conditions actually used by people, known as the *functional stimulus*. Traditionally paired-associates were used to describe the stimulus as presented by the experimenter, or the *nominal stimulus*.

Engelbeck (1986) provided support for Waern's approach to forgetting. Engelbeck represented the knowledge needed to perform utility tasks as a set of production rules. The number of times each production was needed in performing the task was recorded and each production was classified as either a high-frequency competing production, a low-frequency competing production, or an independent production. These two variables (frequency of use and classification) accounted for 80 percent of the variance in the number of observed retention errors made by participants. While the model was able to give a good account for retention, it was unable to provide a satisfactory account for the performance observed during training. This is a serious shortcoming since retention

predictions were based on representations of knowledge assumed to have been acquired during training.

One reason the model may have been unable to account for training performance was because it did not incorporate a detailed process model for procedural acquisition. Following previous studies (Polson and Kieras, 1984; Polson, Muncher, and Engelbeck, 1986; Polson 1988) Engelbeck measured the complexity of training by counting the number of new production rules needed to perform a task. This may have been too great a simplification since, unlike previous studies, the difficulty of acquiring procedural knowledge was specifically manipulated in this study.

Polson and Kieras and their colleagues (Polson and Kieras, 1984; Polson, Muncher, and Engelbeck, 1986; Polson 1988) have used production rule counts to predict training with great success. However, this analysis rest on a tenuous assumption: all productions take an equal amount of time to learn. A count of productions assumes that the amount of knowledge required to perform a task is proportional to how difficult that knowledge is to acquire. To see this, imagine an experiment in which participants learn to log onto a computer system for the first time by typing their user name and a seven character password. Divide the participants into two groups. Give the first group their user name and password on a slip of paper. Next, give the second group their user name only and tell them they must guess their password. We would not expect these groups to take the same amount of time learning to log in. Clearly, the task of discovering their password will slow down the second group. Yet the procedural knowledge needed to perform the task is identical.

The Present Work

The present work is an outgrowth of the work of Engelbeck (1986). Its purpose is to explain two phenomena of great interest in the field of human computer interaction and cognitive psychology: the learning and retention of procedural skill. It attempts to show that representations formed during acquisition can be used to explain appropriate or inappropriate transfer of procedures to new contexts and to explain why some procedures are remembered and others are forgotten. These explanations are given in the form of a computer simulation model. This model attempts to account for the initial acquisition and retention of knowledge needed to perform utility tasks using a menu-based computer system.

The major contribution of the model lies in the integration of components taken from current models of learning and retention. The model constructs the representations assumed by Engelbeck (1986) by storing the results of problem solving episodes in memory.

The model starts with a set of general problem solving methods needed to initially solve the menu tasks. The model learns by generalizing and storing the results of problem solving episodes in an associative memory. The simulation model predicts that procedural knowledge should be difficult to retrieve from memory when that knowledge is poorly connected to other knowledge, or

when the cues needed to retrieve that knowledge are unavailable or no longer make relevant discriminations.

Outline

This work studies skill acquisition and retention in the context of a menu-based computer system. Chapter II gives a short introduction to menu systems. This chapter explains why menu systems are commonly used as interfaces to interactive computer systems. It gives a short description of what menu systems are and how they work. This is followed by a characterization of how people learn to perform tasks using menu systems, the difficulties they have learning to use a menu system, and the problems they have remembering how to perform tasks using the system.

Chapter III describes the simulation model used to predict and explain the performance of participants in the experiments. This chapter starts with a characterization of the model's approach to skill acquisition. The components making up the simulation are described: the domain-general problem solving methods needed for the initial performance of the menu tasks; the mechanism by which the model extrapolates from experience; and long-term memory. An example of the model's acquisition of a task is presented and the model's predictors of performance are described.

Chapter IV presents a re-analysis of a study first reported by Polson, Muncher, and Engelbeck (1986). This study was designed to test the acquisition and transfer assumptions of the model of Cognitive Complexity Theory of Kieras and Polson (1985). This study will be used to test the transfer assumption of the computer model proposed in this work. In addition, since Cognitive Complexity Theory has been able to give very good fits to these data, it provides an opportunity to compare the proposed model directly with Cognitive Complexity Theory.

Chapter V describes a new empirical study designed to test the transfer and retention assumptions of the model. This is based on a classical negative-transfer study. It examines the effects of well-cued and poorly-cued inconsistent methods on the acquisition and retention of procedures. Inconsistencies arise when different methods are required to accomplish similar goals. To accomplish successfully tasks employing these methods, people must discriminate when and where different methods are appropriate. These inconsistencies can produce large interference effects.

The last chapter concludes the dissertation by summarizing the findings of this work. These findings are reviewed and their implications for cognitive psychologists, educators, and system designers are discussed. Finally, the shortcomings of this work are discussed and directions for future work are suggested.

CHAPTER II
MENU-BASED COMPUTER SYSTEMS

Before the model and experiments are presented, a short review of menu systems is given to familiarize readers with the domain, motivate the selection of menu systems as the domain of study, and review difficulties encountered by users learning and remembering how to perform tasks with these systems.

What Are Menu-Based Computer Systems?

Menu selection is one of the most common methods used to structure people's interactions with computer systems. Schneiderman (1987, page 86) gives four reasons why menu systems have been popular with designers: 1) Menu systems can eliminate training; 2) They can eliminate memorization of complex commands; 3) Users can indicate their choice with a few key presses; and 4) They can be used effectively by both experts and inexperienced users, if properly designed.

As Schneiderman's list indicates, menu systems are relatively easy for people to learn and remember. Like restaurant menus, computer-based menu systems present their users with a list of choices. To perform tasks with these systems, users select items from the presented list. One of three things typically happens when a selection is made: First, code may be executed, performing some task; second, the system may make a transition, presenting a new menu to users containing a more refined set of choices; or third, the system may prompt for the parameters needed by its underlying routines. The performance of a task usually requires a series of menu selections and the entry of several parameters needed by the system.

Figure 2-1 shows a menu from the system used in this study. It is typical of many menus, containing a great deal of information to guide users: a menu title, menu items, and prompts. The title helps distinguish this particular menu from other menus and to indicate what type of tasks are accessible from the menu. The menu items ("check document", "clear supplement", and "load supplement") provide specific task cues that indicate the tasks that can that performed from this menu; documents can be checked for spelling errors, spelling supplements can loaded or deleted. At the bottom of the menu, prompts give instructions for selecting or performing a task.

Why Study Menu-Based Systems?

There are three reasons why menu systems were chosen as a vehicle for studying the learning and retention of procedural knowledge. First, learning to perform menu tasks is not difficult. Perhaps the easiest way to learn how to do something is by being told. Generally, system prompts tell inexperienced users what can be done and how to do it. Since most

inexperienced users have experience following instructions, they can apply this knowledge to the menu tasks facing them. This has two benefits: Users learn to perform tasks in a reasonably short period of time and the problem-solving knowledge needed to simulate initial learning of menu tasks is a reasonably constrained subset of people's problem-solving knowledge.

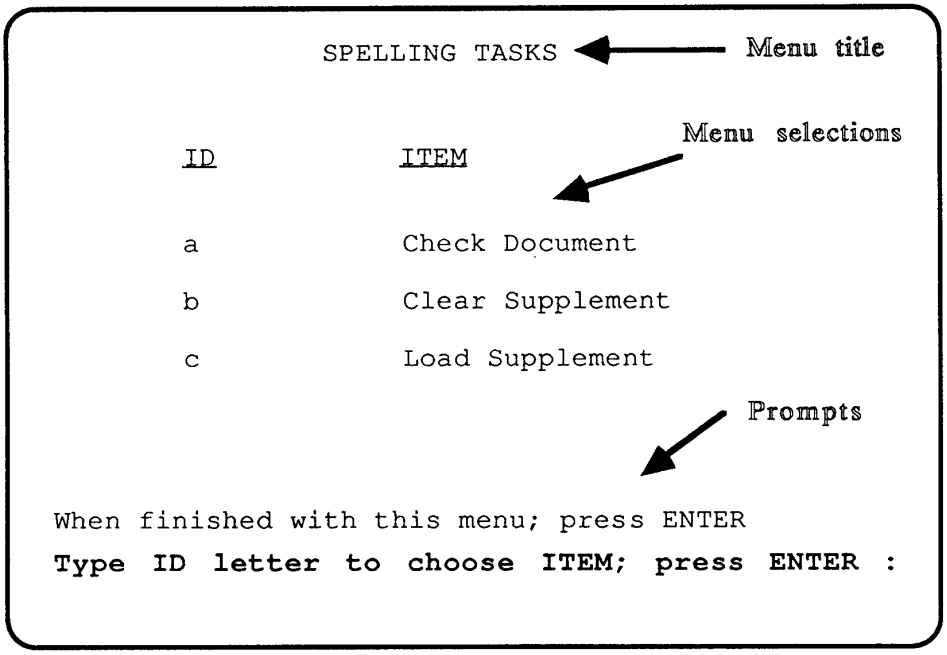

Figure 2-1. Example menu

The second reason for studying menu systems is pragmatic. Since menu systems are in wide use, and will be for some time, any knowledge that can make them easier to learn and remember is important. The results provide insights into how people initially learn to use a menu system, how they use their prior experiences as guides for action in new contexts, and how the representations built during learning determine later recall. This knowledge can be used to improve menu systems.

Finally, menu tasks bear many similarities to tasks using verbal materials. Menu tasks are procedural tasks performed as a series of discrete steps. At each menu-task step users must execute an appropriate method to move onto the next step. Each task step is associated with a system state (e.g., current menu, system prompt, and highlighted item) which can be used to determine when a particular method is appropriate. The state of the menu system and the task can be considered as cues that users learn to associate with methods. Given this characterization, it should be possible to identify many cues attended to by users during learning. It should be possible to determine the cues on the menu that serve as the functional stimuli. A number of researchers have argued that the discrimination of the correct response from the set of possible

responses is the "retention bottleneck" in procedural tasks (Hurlock and Montague, 1982; Ericsson and Crutcher, 1988).

Difficulties People Have Learning Menu Systems

Although menu systems may be easier to learn than other types of interfaces, they are not trivially easy to learn and remember. People must acquire a great deal of knowledge to perform tasks using a menu system and they can encounter a number of difficulties using these systems. The difficulties inexperienced users encounter learning to use a menu system include: 1) following instructions; 2) learning when instructions are inappropriate; 3) making correspondences between menu items and task descriptions; 4) generalizing from past experiences; and 5) evaluating whether actions are leading to a solution. Unfortunately, empirical studies of menu systems have only addressed the second of these issues: difficulties users have in making correspondences between menu items and task descriptions (See, Schneiderman, 1987 for a summary of many of the current empirical issues and studies).

Following Instructions

Following the instructions would appear to be a simple and straightforward part of the user's task. Yet, anyone who has had to put together devices that "require some assembly" can attest to the perils of following instructions. A user must comprehend the instruction and then develop a procedure for carrying out the designated action. Instructions seldom provide a list of concrete user-actions (e.g., press the "d" key; press the ENTER key). Instead, instructions are usually abstract, referring to user goals and system concepts (e.g., "Type ID letter to choose ITEM; press ENTER"). The difficulty is not so much following instructions, but understanding them in the first place. Anderson (1987a) reports that people have difficulty integrating instructions into ongoing problem-solving efforts, and have difficulty understanding abstract instructions. These difficulties occur when people learn to use menu systems.

The prompts given by menu systems can require a great deal of effort to translate into procedures. The prompt that appears at the bottom of the screen in Figure 2-1 reads "Type ID letter to choose ITEM; press ENTER". To create a correct procedure from this prompt, several correspondences must be made between the prompt and the menu, and between the menu items and task goals. A correspondence between the ID and ITEM of the prompt and the ID and ITEM columns of the menu must be made. Next, the ITEM appropriate for the current task must be recognized and the correspondence between this ITEM an ID letter made. Finally, the chosen menu item must be typed and the ENTER key must be pressed. The failure of any one of these steps can lead to a error.

Learning When Instructions Are Inappropriate

An additional difficulty facing users is learning when system instructions are inappropriate. When users learn to perform a menu task, part of what they must learn is when the system's prompts should be followed. For instance, using the menu system in this study

users must make a number of menu selections to get to a terminal menu from which an action can be performed (see Figure 1). To complete a task, users must perform some actions, such as changing a menu parameter and then leave the menu. Users must leave the menu because the completion of the task does not result in their exiting from the menu. Since a number of tasks can be performed at most terminal menus, and the designers thought users might want to perform multiple tasks, the designers had the system leave users at the terminal menu. The designers then developed an alternative method to allow users to leave the menu: The typical way to leave the menu is by pressing the "ENTER" key. This meant that when users wanted to leave the menu, the main prompt instructed them to make a menu selection. Inexperienced users have difficulty learning to leave the menu by ignoring the main prompt and pressing the ENTER key.

Making Correspondences Between Menu Items and Task Descriptions

Often menu choices can be difficult for inexperienced users to select: They may not understand what an item means (e.g., "paginate"); they may not recognize a menu item that is a synonym for the task they wish to perform (e.g., "duplicate" versus "copy"); two or more menu items may appear to correspond to a desired task (e.g., "change default document format" versus "change alternate document format"); or they may be forced to choose between menu items that represent categories they do not understand (e.g., "work diskette tasks" versus "program diskette tasks").

When users and designers do not agree on how a domain should be partitioned, users can have difficulty navigating through the menus. Dumais and Landauer (1983, cited in Furnas, Landauer, Gomez, and Dumais, 1984) found that people match category members (e.g., typewriters) to their correct category description (e.g., products you get and take away with you) on average 50 percent of the time. Schwartz and Norman (1986) relabeled top-level menu items for CompuServe making the menu items more distinctive from each other[1]. Making the items more distinctive resulted in better performance for finding menu items on lower level menus than did the original CompuServe menu items.

Extrapolating From Past Experiences

People commonly use their past experiences to guide their actions in a new context. If successful, an extrapolation from a past experience can reduce the amount of search required to find the appropriate action. Successful extrapolations can reduce the knowledge people need to use a system. Yet, extrapolations from experience, like all inductive inferences, can result in errors that hinder performance.

Interface designers are sensitive to this issue and have emphasized consistency as one of the major characteristics of a well-designed interface. Reisner (1981) has proposed a cognitive grammar which is used, in part, to check the consistency of a systems' interaction with users. Reisner (1981, page 234) states that consistency occurs when "tasks or subtasks that are

[1]Unfortunately Schwartz and Norman (1986) used their own judgements of distinctiveness to make the items more distinctive.

perceived as similar by the user are described by similar sequences of actions." She gives the past tense of English verbs as an example of structurally inconsistent rules. A simple rule for forming the past tense of English verbs is to add "ed" to the end of the verb, producing the past tense form. However, this is not an adequate rule for deriving the past tense of all English verbs (for example, "go" --> "goed"). Additional rules are needed to remedy this problem. These additional rules make the derivation of the past tense of verbs inconsistent since the derivation requires more than a single necessary rule.

Computer-user errors are often explained by referring to a system's inconsistent interactions with users: The user tried to leave the menu but the method was inconsistent with other methods used; this inconsistency lead to the error. Inconsistencies are design flaws diagnosable from syntactic descriptions of a system's interactions with users.

Evaluating the Results of Actions

Lewis and his colleagues (Lewis and Mack, 1982; Mack, Lewis, and Carroll, 1983) have shown that users have difficulty recognizing when their actions have led to an error. Lewis and colleagues report that users frequently misinterpret system responses to their actions and readily give explanations for a systems' behavior in terms of the task they are performing. Because these users are able to explain away their errors, they perform more errors. The result is that they usually make still more errors, so that recovery from their original error is difficult and at times impossible.

Summary

The present chapter presented a short review of menu-based computer systems. Its purpose was to provide an overview of the difficulties people have using these applications and to motivate the use of menu systems to study initial skill learning and retention. A number of difficulties people have learning and remembering how to use menu systems were reviewed: 1) difficulties following the instructions given by prompts; 2) difficulties discriminating when actions are appropriate; 3) difficulties making correspondences between their goals and menu descriptions; 4) difficulties evaluating the results of their actions; and 5) difficulties extrapolating past experiences to new situations.

The difficulties users have with menu systems show that they are not trivially easy to learn and use. But they are easy to learn and use when compared to many other computer applications. Many people are able acquire the knowledge needed to perform menu tasks competently in a matter of hours. Indeed, their relative ease of use has made menu systems a popular interface style for computer applications. Menu-based applications provide a great deal of information to their users: Prompts tell users what to do, while menus tell them what tasks can be performed. The information provided serves to constrain the amount of search needed to perform menu tasks. Because of this constrained search space and the discrete nature of menu tasks, menu systems provide a good domain for investigating the learning and retention of procedural knowledge.

CHAPTER III
MUDDLE: A MODEL OF LEARNING AND RETENTION

The present chapter introduces the Muddle model. Muddle simulates the learning and retention of procedures needed to perform menu-based word-processor tasks, although the principles on which it is based are presumably involved in most, if not all, human skill acquisition. By accounting for both the learning and retention of procedures, the model should provide insights into the relationships between these two phenomena.

The chapter begins with a characterization of how people approach the task of learning everyday skills. Next, similar and contrasting views of what is learned from problem-solving episodes are described. This is followed by a description of the model as a cognitive architecture, which leads to an example of the model learning a menu task . The discussion ends with a description of the data Muddle is meant to account for and how its predictions are derived.

Muddling

In the introductory chapter it was pointed out that people have understandably little interest in many of the everyday skills they must perform. Few people are interested in *how* automated tellers are used to withdraw money. Instead, their interest is with withdrawing the money itself. Perhaps as a consequence of their lack of interest in the mundane details of these tasks, people approach them in a haphazard way describable as "muddling".

Consider as an example a first-time user working his or her way through a task using a menu-based word processor . Say this novice user wishes to check a document for spelling errors and faces the spelling-tasks menu shown in Figure 3-1. Assuming that this user, like many others, has previous experience using automated machines and services, he or she is likely to search the menu for instructions. Two prompts can be found at the bottom of the spelling-task menu. The first prompt directs users to press the ENTER key when finished with this menu. Since there is no clear connection between checking spelling and being finished with the menu, we can assume the novice will abandon this prompt and turn his or her attention to the second prompt. This prompt directs users to "Type ID letter to choose ITEM; press ENTER". At this point the user probably does not know what an ID-letter is. An examination of the menu revels a column of three letters (a, b, and c) headed by the label ID. Of course these letters have no clear connection to the novice's goal of checking the spelling on a document. However, these ID-letters correspond to ITEMS on the menu that might be related to the users goal. The user looks at the ITEM column of the menu searching for an item that corresponds to "check spelling". None of the menu items mentions spelling. However, item "a", check document, is a close match to the goal. The novice, therefore, decides correctly to type "a" and press the ENTER key. These actions change the state of the system, hurling the novice onto the next task step.

This example illustrates how weak problem solving heuristics guide behavior. What got the novice through the step was knowledge of how to follow the instructions provided by the prompt, an ability to make correspondence between various things on the menu, and judgments of similarity between item descriptions and goals. These weak heuristics suffice for this tasks and for many others. Although the behavior of the user lacks the fluidity of an expert at the tasks, the user managed to accomplish meaningful work without attending training courses or reading manuals.

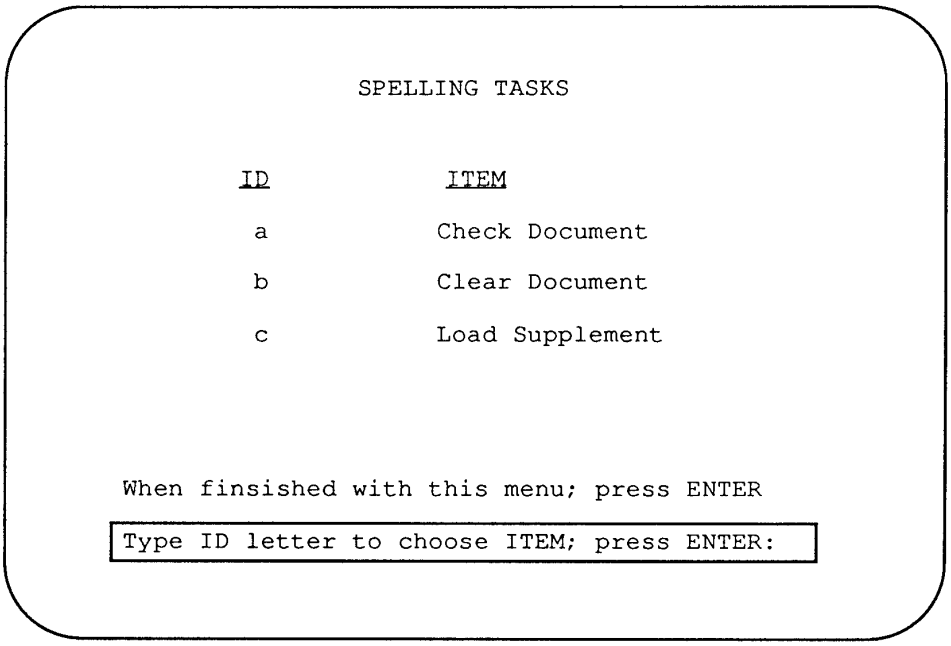

Figure 3-1. The spelling-tasks menu.

What has the novice in the example learned from his or her experience with the menu system? Intuitively this may appear obvious, but it is not. Problem solving and learning are different things: One can solve a problem without learning anything. Conventional computer programs are examples of things that solve problems without learning anything. Learning occurs by encoding problem solving experiences in such a way that they can be used to solve the original problem and new problems. What could the novice user have encoded about his or her encounter with the menu system?

The novice might have encoded that the first step required to check a document for spelling errors is to type "a" and press ENTER. This assumes that users encode their experiences in terms of task goals. Under this view, the interaction with the spelling task menu was just one subgoal on the solution path that must be taken to check a document for spelling errors. Another possibility is that the novice encoded the knowledge that following the type-ID

prompt was appropriate and how this was done. Here the novice encoded what actions were appropriate and in what system context they should be performed.

These two views of encoding problem-solving experiences differ on the roles played by goal state and system state information. The first view is a *task-centered* view of encoding. It states that users encode their procedures in terms of task goals; under this view, goal states dominate representations of procedures. The second view is a *system-centered* view which states that people encode their knowledge in terms of system states and that system states come to dominate procedural representations. It would be wrong to conclude that these two views are mutually exclusive. Most models mix goal state and system state information. The purpose of this dichotomy is to classify existing models in terms of the emphasis they give to goal and system state knowledge.

There are two important implications of the system-centered view of encoding. The first is that the system context should play a dominant role in controlling the behavior of novices. System information determines what problem solving methods will be used to guide behavior. The system context will also provide the majority of the cues needed to direct the selection of domain-specific procedures as experience with a system increases. The second implication is that people should aggressively transfer their knowledge when the system context is similar across tasks. Since little knowledge about specific tasks is encoded, the selection of actions will be based mainly on available system information. The system context alone does not always tell the user *what* action to take, only *what type* of action to take (e.g., type an ID-letter, provide a document name).

The Muddle model implements a strong form of the system-centered view. It assumes people will use the system context and their knowledge of previous actions to select their next action.

Existing Characterizations of Learning

The model introduced in this chapter was christened Muddle because muddling; progressing in a haphazard way seemed a fair characterization of how many people, including the participants in the studies reported later, approach learning computer applications. Of course, other researchers have noticed the haphazard way people go about learning computer applications.

Muddling is related to the concept of *active learning* described by Carroll and his colleagues (Lewis and Mack, 1983; Mack, Carroll, and Lewis, 1984; Carroll and Rosson, 1987). Carroll and Rosson (1987) identify *production bias* as a key property of active learning. Production bias is the preference learners have for accomplishing tasks rather than understanding the instruments being used to accomplish those tasks. They prefer to use what they already know works, instead of learning a more effective way to perform a task.

This characterization of inexperienced users as active learners suggests a general control structure for learning. Inexperienced users prefer to apply known methods before trying to generalize existing knowledge (assimilate) or acquire new knowledge through problem solving.

They prefer to generalize existing knowledge before engaging in problem solving to acquire new knowledge. As a last resort, they use what the system provides to learn what they do not know and could not assimilate. This characterization of learning is similar to the basic control of the Muddle model.

Muddling is also related to the concept of *situated actions* given by Suchman (1987). Suchman has criticized what she calls the "planning model of action". The planning model assumes that human action is organized by highly structured plans that are used to synthesize the procedures needed to perform actions. She argues that the planning model is insensitive to changes in the environment and suggest that human actions should be viewed as *situated actions*: actions chosen with reference to the environment in which they take place. Suchman cites the study of Feitelson and Stefik (1977) as an example of how actions are based on situations. Feitelson and Stefik observed geneticists planning their experiments. Feitelson and Stefik found that geneticists did not plan their experiments by decided what sequence of experiments to perform and then performing them. Instead, the geneticists deciding what experiment to perform next by relating their recent results to global research goals.

Suchman's distinction between planned and situated actions resembles the distinction made earlier between task-centered and system-centered representations of procedures. Like situated actions, the system-center view of encoding depends on the environment to guide behavior. Alternatively, planned actions resemble the task-centered view of encoding. The task-centered view uses a hierarchical goal structure to control behavior. This encoded goal structure is a plan for achieving a task.

Finally, another characterization of the system-centered view has been fostered by grammatical notations used to represent the interactions of users with computer systems. Several notations have been suggested to describe possible user dialogs with a system: Backus-Naur Form (BNF) or extensions to BNF grammars (Hanau and Lenorowitz, 1980; Reisner, 1981; Schniederman, 1982), and transition diagrams (Parnas, 1969; Wasserman and Stinson, 1979; Jacob, 1985).

Initially, grammatical formalisms were used to describe how users interacted with systems, or "how to use it" knowledge. But Reisner (1981) raised the status of these grammars by making claims for the psychological validity of grammatical representations with her cognitive grammar. Reisner used her cognitive grammar to make psychological comparisons between two alternatives designs for a graphics editor. Using characteristics of the grammatical representation of the systems she was able to predict learning time, efficiency of interactions, and errors made during learning and retention.

Like the other grammatical formalisms, Reisner's cognitive grammar represented *how* users interacted with a system. Her cognitive grammar makes the assumption that users will represent their procedural knowledge in terms of how the system worked, rather than how one performs tasks using the system. This system-centered representation is another embodiment of the system-centered representation assumed by the Muddle model.

Of course, not all researchers accept the system-centered view. The most well-known

model advocating a task-centered encoding of procedures is Cognitive Complexity Theory (Kieras and Polson, 1985). Cognitive Complexity Theory (CCT) incorporates the assumptions that the GOMS model of Card, Moran, and Newell (1980; 1983) makes about how people structure their knowledge. The GOMS model assumes that the knowledge of users can be represented in terms of *goals*, *operators*, *methods*, and *selection rules*.

Goals represent the intentions of users to perform tasks, subtasks, and cognitive operations. The GOMS model organizes its knowledge into hierarchical goal structures, which in turn are used to control the sequencing of methods and operators.

Operators in the model correspond to primitive mental and physical actions, such as creating goals and pressing keys. Methods are essentially subroutines for generating complex sequences of actions. Examples of methods include: 1) making a menu choice; 2) deleting a word from a prompt line; and 3) filling in a menu parameter. Selection rules, the final category of knowledge, choose which method from a set of possible methods should be executed. Selection rules choose methods based on examinations of goals and system context.

CCT uses the representational assumptions of GOMS to model the knowledge needed by novice users to perform computer applications. Using production systems to represent the knowledge described by the GOMS model, Kieras, Polson and their colleagues have be able to give very good accounts of the behavior of novices learning to perform tasks using computer applications (Polson, Kieras, Engelbeck, and Willer, 1984; Polson and Kieras, 1984; Polson, Muncher, and Engelbeck, 1986; Polson, Bovair, and Kieras, 1987). CCT's acceptance of the strict hierarchical structure of knowledge yields representations of knowledge that are dominated by task goals. This is because GOMS models skill performance rather than skill learning. GOMS represents the procedures that are thought to be the product of a learning mechanism. The result is domain-specific procedures requiring little or no search. These procedures use task-specific goal structures to control performance.

Other theories of skill attempt to model skill learning and skilled performance. These models rely on problem solving knowledge and learning mechanisms to structure domain-specific knowledge. The structure of Muddle is based on these cognitive architectures.

Cognitive Architecture

Muddle draws heavily on two well-known general theories of cognition, ACT* (Anderson, 1982; 1983; 1987a) and SOAR (Laird, Rosenbloom, and Newell, 1986; 1987). These two models are briefly reviewed in preparation for a discussion of Muddle. This review serves two purposes. The first is to recognize the contributions of these two models to the Muddle model. Secondly, this description puts the Muddle model into perspective by comparing its basic assumptions with those of ACT* and SOAR.

What is a Cognitive Architecture?

Anderson (1983, page ix) states that a cognitive architecture is "a theory of the basic principles of operation built into the cognitive system". A central issue for cognitive architectures

is what controls the transition from one mental action to another.

SOAR (Laird, Rosenbloom, and Newell, 1986; Laird, Newell, and Rosenbloom, 1987) and ACT* (Anderson, 1983; 1986; 1987a) have been put forward as cognitive architectures. SOAR and ACT* differ in their scope. ACT* is intended to model human cognition, while SOAR only provides an architecture for problem solving and learning by experience. Parts of ACT* relevant to this work are implemented in the production system language GRAPES (Goal-Restricted Production System) (Sauers and Farrell, 1982).

SOAR and ACT* share many important architectural features: 1) They both represent procedural knowledge as productions; 2) Both are problem-space architectures - that is, both view problem solving as the basic unit of cognition; 3) They use hierarchical goal structures to control behavior; 4) They employ learning mechanisms which work by analyzing and storing the results of problem-solving episodes. The basic claim of made by ACT*'s and SOAR's use of productions is that a set of condition-action pairs underlies cognition. This assumption represents a modern form of associationism.

Production System Representation

Anderson (1983) advocates the use of production systems as the "general computation architecture" for a theory of mind. Production systems can generally be separated into three components: 1) a set of production rules describing procedural knowledge; 2) a working memory containing a description of immediate goals, intermediate results of computations, and representations of the state of the world; and 3) an interpreter that controls the selection of the production rules.

Each production rule associates a set of conditions with a set of responses. The conditions must be satisfied before the actions of the rule can be executed. The action component is executed if the conditions of the production are satisfied and it is chosen by the interpreter for execution.

Production system operations can be divided into a sequence of *recognize-act* cycles. During each recognize-act cycle the interpreter selects and executes a production rule. In the recognize part of the cycle the conditions of each production rule are matched against the contents of working memory. Rules that match the contents of working memory are then ordered by a conflict resolution mechanism. The dominant rule is then executed in the act part of the cycle. This action can include physical and/or cognitive components. Once the actions of the selected production rule are executed the system again enters the recognize phase of the recognize-act cycle.

Productions are used to represent procedural knowledge for several reasons. Productions provide a well-structured, simple, homogeneous representation. Productions can be made independent of each other, making it possible to add and retract productions without greatly disturbing a system. The structure and independence of productions aids in the building of an incremental learning mechanism.

Problem Solving

Both SOAR and ACT* assume that the basic category of behavior is problem solving. Newell's (1980) problem space hypothesis provides a strong statement of this belief in his *problem space hypothesis*. The problem space hypothesis asserts that the fundamental organizational unit of all human goal-oriented symbolic activity is the problem space. A problem space is a space of symbolic states representing the possible domain states. Navigation through the domain is accomplished by the application of operators that allow a problem-solver to get from one state to another. Under this view, problem solving can be characterized as search in the problem space where the problem solver must find a path, produced by a sequence of operators, that takes him or her from an initial state to a goal state.

Although search in a problem space has been used mainly to describe heuristic problem-solving behavior, the problem space approach can also incorporate of algorithmic solutions to problems. Algorithmic solutions can be seen as search where search-control knowledge is sufficient to completely constrain search. Within the problem-space, it is possible to generate a continuum starting with heuristic search, where little search-control knowledge is available, to algorithmic search, where a great deal of search-control knowledge exists.

Problem-solving knowledge in problem-space architectures can be divided into two categories: control knowledge, and domain-specific knowledge (i.e., knowledge of goals, states, and operators). The separation between control knowledge and domain-specific knowledge has been suggested by several researchers (Davis, 1980; Weyhraurch, 1980; Genesereth, 1983). This separation allows the control of behavior to be partially specified and makes control knowledge open to inspection. Control knowledge can be incompletely specified during problem-solving. For example, if control knowledge cannot be used to decide which of several operators should be applied to a state, one can be chosen randomly allowing problem solving to continue. Later, if search-control knowledge is added which allows the system to choose the best operator, the system's performance on the problem can be improved. The ability to inspect and modify control knowledge allows operators to be chosen more intelligently. For example, control operators can be re-ordered to prevent unnecessary operations from being performed.

Problem solving allows a system to perform in domains where little search control knowledge is known. A large number of methods have been developed that organize search and make few demands on domain knowledge. These methods are known as weak methods. Example weak methods are: depth first search, hill climbing, operator subgoaling, and means ends analysis. Anderson (1987a) gives an example method for steepest ascent hill climbing represented as a production:

IF the goal is to transform the current state into the goal state
THEN set as subgoals
1) To find the largest difference between the current state and the goal state.
2) To find an operator to eliminate that difference.
3) To convert the state that results from the application of this operator to the goal state.

This example contains search-control knowledge without reference to domain-specific knowledge. The control knowledge decomposes the problem into subproblems by spawning subgoals prioritized by order. These goals are an important input to the learning mechanism of both ACT* and SOAR and will be discussed in detail later. The conditions of the productions are used to determine the application of a specific weak method. To solve a problem the conditions of a production system must either be capable of performing actions that will not fail, or the problem-solving system must be able to recover from failures. Since problem solving by its very nature is prone to failure, both ACT* and SOAR use additional problem-solving to recover from failed problem-solving attempts.

Learning in a Problem Space Architecture

Domain-specific knowledge is acquired through problem solving. In this type of architecture, problem solving and learning are intricately connected. Learning takes the problem solver from the state where little search-control knowledge is available to the point where extensive search control knowledge is available. The learning mechanism interprets the results of problem solving episodes, making decisions about what can be learned from these episodes. The analysis permits the extraction of control knowledge that can reduce the amount of search that must be performed in the future.

Learning mechanisms improve task performance in many ways. One way is to improve the efficiency of search. Several methods for improving efficiency are to add goals and domain knowledge that constrain search, explore paths based on the frequency of success (ACT* does this through a production strength mechanism), or learn when to abandon search. Caching mechanisms are the basic learning mechanisms used by ACT* and SOAR. Caching mechanisms improve performance by analyzing the results of problem solving episodes and storing these results as domain knowledge. Another way to improve performance is to restructure problem-solving procedures (c.f., Neches and Hayes, 1978; Lewis, 1981).

Learning in ACT* occurs through a process called knowledge compilation (Anderson, 1986; 1987a) that analyzes a trace of executed productions. The compilation process can be divided into two components: proceduralization and composition. Proceduralization is a process by which domain-specific productions are created from domain-general productions. Compilation serves to collapse sequences of productions into a single production that accomplishes the actions of the sequence.

ACT*'s proceduralization mechanism builds a domain-specific production from a general production by modifying a copy of the general production. The modification process takes a copy of the general production with the instantiation of its variables. Conditions that reference the domain context are eliminated from the copy. The remaining goal condition and actions of the production are then made domain-specific by replacing variables with their instantiations. Anderson (1983) gives the creation of a domain-specific production for dialing a phone number as an example of this process. The domain-general production states that in order to dial someone's telephone number, you dial the digits in their number. Here the variables refer to

who that "someone" is and what the "digits in their number" are. When faced with a new task of dialing the phone number for a specific person, the name of the person (Mary) and her phone number (432-2815) are bound to the variables in the general rule giving a specific procedure for dialing Mary's number. The bound version of the production is added to memory completing the proceduralization process. This production improves performance because it no longer requires the maintenance of the phone number in working memory, unlike the general production. Performance is further improved by composition, which collapses the actions of several rules into a single rule. Procedural knowledge can also be tuned through a strengthening mechanism.

The composition process collapses over sections of a goal tree to create a single production that duplicates the same behavior as the sequence of productions (c.f., Lewis, 1982). Composition acts on domain-specific productions by first collecting the conditions of the productions that are to be composed The conditions are then inspected to determine which need to be included in the condition side of the composed rule. There are a two rules for determining which conditions will be included in the composed production: 1) A condition that tests the contents of working memory is included, unless it tests a condition that is placed in working memory by a production that is to be composed; 2) A test for a goal is included in the composed rule, unless the tested goal is one that is created by another production in the group being composed. The actions included in the composed rule include all insertions made into working memory by the sequence of productions, unless the action spawns one of the composed subgoals.

Learning in SOAR occurs through a process called chunking (Laird, Rosenbloom and Newell, 1986, Laird, Newell and Rosenbloom, 1986). Chunking begins when a subgoal terminates successfully. The chunking mechanism must solve a credit assignment problem to identify the conditions and actions of the chunked productions. The links between objects in the context stack, augmentations, and preferences are analyzed to make these determinations. Actions are defined as the working memory elements created within the subgoal that have links to higher contexts and their elaborations. This definition eliminates working memory elements added to hold intermediate results.

To construct the chunked productions, SOAR keeps a record of the productions that fired during the processing of the subgoal. Only productions that added elements to working memory are saved. Also associated with each production is a record of its associated context element, this keeps a record of the goal context in which the production fired.

When a subgoal successfully terminates, the actions of the subgoal are recalled and divided into groups. Actions that are linked to one another, either directly or indirectly, are put together into a group. Each group forms the actions of one of the chunked productions. Once the actions have been divided up, the conditions of the chunked productions must be determined. Productions that fired during the subgoal and that share an action with an action group are associated with that group. The conditions of these production groups are used in creating a chunked production for that action group. These conditions can be divided into two subgroups, a subgroup of conditions existing before the creation of the subgoal, and a subgroup of

conditions existing after the completion of the subgoal. The prior conditions become the conditions of the chunked production, while those existing afterwards are treated as if they were actions of the subgoal.

Chunking in SOAR has two important characteristics: productions cache the results of problem solving within the subgoal, and, as Laird, Newell and Rosenbloom (1986) claim, the chunked productions are able to transfer easily to other problems since the analysis produces a minimal set of conditions in the chunked production.

In summary, both ACT* and SOAR are cognitive architectures that assume that problem-solving is the fundamental unit of cognition. They describe cognitive behavior as search in a problem space, where behavior in an unfamiliar domain is guided by general problem-solving knowledge. Both assume that learning occurs by analyzing and storing the results of problem-solving episodes in memory as productions.

Muddle as a Cognitive Architecture

Muddle shares many basic architectural assumptions with ACT* and SOAR. Like its predecessors, Muddle views problem-solving as the basic unit of cognition. It, too, learns by storing the results of problem solving episodes in memory. However, Muddle differs from these models in two basic assumptions. First, it represents procedural knowledge in an associative network rather than as productions. Also, Muddle does not rely on hierarchical goal structures to control behavior to the same degree as ACT* and SOAR. These differences stem from the implementation of the system-centered encoding assumptions and the need to model retention.

An associative network was chosen over a production system representation of procedures for two reasons. First, the associative network provides a simple, well-specified account for forgetting as associative interference. A model of memory already existed which assumed knowledge could be represented in a associative network (Raaijmakers and Shiffrin, 1980; 1981). This model can be considered a consensus model of memory and has accounted for an impressive number of memory phenomena. Muddle uses these assumptions to produce an associative interference model of procedural forgetting.

The second reason why an associative network was chosen over a production system representation was the need to have Muddle retrieve actions based on partial matches to system states. In the past, partial matching mechanisms have proven to be notoriously difficult, cumbersome additions to production system models (Kline, 1981). In contrast, the associative net representation performs partial matches without any additional mechanisms. Note that the associative net representation does not violate the basic structure of procedural knowledge where conditions are associated with actions to produce behavior. The associative net representation of Muddle also pairs conditions with actions.

The second major difference between the architectural assumptions of Muddle and those of ACT* and SOAR is the role played by goals. In the past, applications of ACT* and SOAR have relied on goals as the major mechanism for controlling behavior. Muddle does not rely as

heavily on goals. Muddle assumes a system-centered view of encoding where novices structure their procedural knowledge in terms of previous actions and system states.

How Muddle Models Users

This section presents the control structure of the model. The inputs to the model are described, as well as the processes that act upon these inputs to produce behavior.

There are three types of inputs to the model. The first is a representation of task instructions. Task instructions are represented in a propositional format. The task instructions provide the model with a high level goal to solve. The second input is the menu system's state, again represented as a set of propositions. This description includes menus and feedback given by the menu system. The final input is feedback about the correctness of the model's actions. Feedback is crude: the simulation is told whether or not it produced the correct answer.

Figure 3-2 divides the simulation into two components: a component that simulates and maintains the representation of the environment, and a problem solving component, Muddle, which evaluates the current state and selects operators to apply to that state. The result is a read-evaluation-print loop in which the current problem state is read into the problem solver and is evaluated. An operator is then selected that changes the current state.

Table 3-1 provides a sample of the task instructions given to participants for checking a document for spelling errors and their propositional representation. The top half of the table shows the instructions as they appeared to participants, while the bottom half shows the corresponding propositional representation provided to the model. The first proposition describes the goal of the task: check document "intro" for spelling errors. The next two propositions provide general information contained the instructions; that "intro" is a document and that "paper" is a diskette. The final proposition states the relationship that exists between the document and diskette: document "intro" is on diskette "paper".

The state of the menu system is also represented as propositions. Table 3-2 shows the propositional representation of the "Spelling Tasks" menu shown in Figure 3-1. These propositions describe the parts of the menu (title, items, id letters, prompts, and messages), and information about the state of the menu (whether any menu items are highlighted, and what characters, if any, appear on the prompt line). The first proposition in Table 3-2 states the title of the menu. Propositions two, three, and four list the three ID letters appearing on the menu, while propositions five, six, and seven list the corresponding menu items. The next three propositions represent the correspondences between the ID-letters and the ITEMs on the menu. On the real menu this information is conveyed by the spacial configuration. The final two propositions describe the two system prompts. Proposition eleven describes the systems' secondary prompt instructing users to press the ENTER when they have finished with the menu. Proposition twelve is a description of the main system prompt instructing users to choose an ID letter, then press the ENTER key. The proposition does not represent all the information contained in the real prompt. The description of *why* an ID-letter was to be typed (i.e., to

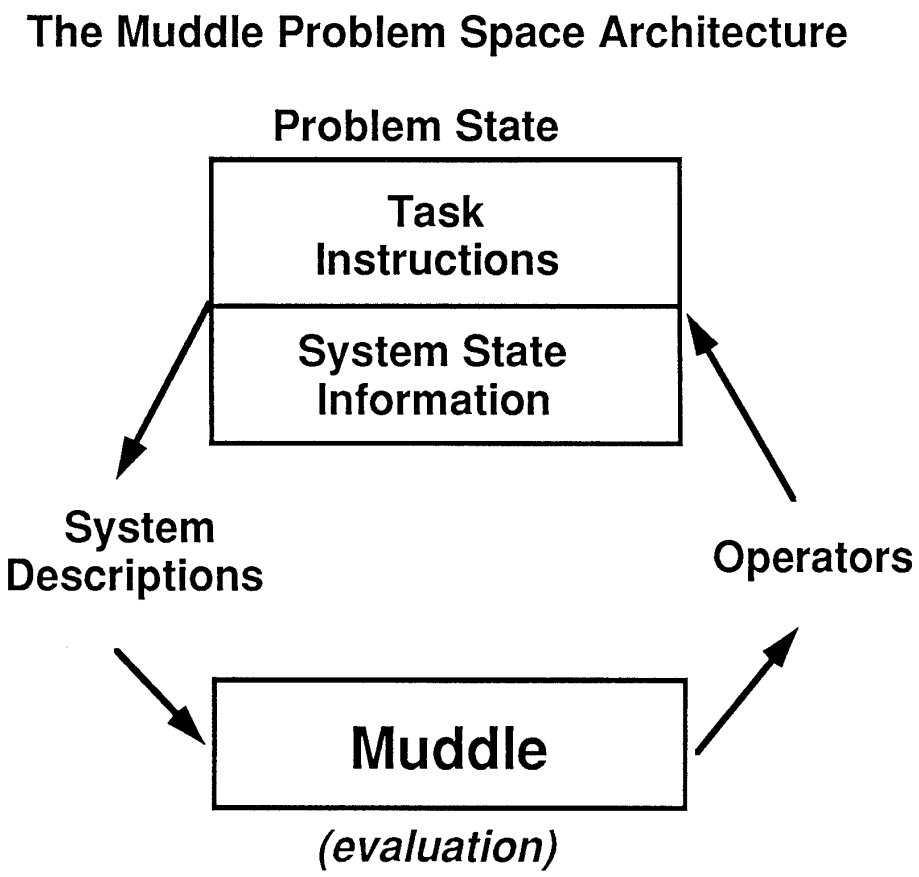

Figure 3-2. Diagram of Muddle's architecture.

choose an item) was left out of the description because it was not needed and would have required additional problem solving to interpret and then ignore this part of the prompt.

The propositional representation of the spelling-tasks menu only represents how the top-level menu would initially appear to the users. As the model performs actions (typing an ID letter followed by ENTER), the menu system changes state and a new menu must be displayed to Muddle. These changes are made by special-purpose code associated with each menu. This code simulates the behavior of themenu system, changing the representation of the system in response to the actions of the model. These changes reflect the display of new menus and prompts, the appearance of characters on the prompt line, highlighting of menu items, and error and system messages that are presented. In addition to managing the description of the system state, this special-purpose code assures that Muddle's procedural knowledge can adequately describe interactions with the application.

Table 3-1.
Instructions For the Check Spelling Task Given To Experimental Participants and Their Corresponding Propositional Representation of the Instructions Used by the Model

Instructions Given To Participants
Check the document **intro** on diskette **paper** for spelling errors.
Propositional Representation Used by the Muddle Model
goal(check(spelling(document(intro)))). document(intro). diskette(paper). isOn(document(intro),diskette(paper)).

Table 3-2.
Propositional Representation of the Spelling-Tasks Menu Used by the Muddle Model

Number	Proposition
1	menuTitle(spellingTasks).
2	id(a).
3	id(b).
4	id(c).
5	item(check(spelling(document))).
6	item(load(supplement)).
7	item(clear(supplement)).
8	corresponds(id(a),item(check(spelling(document)))).
9	corresponds(id(b),item(load(supplement))).
10	corresponds(id(c),item(clear(supplement))).
11	secondaryPrompt((whenFinished(menu),press(enter))).
12	mainPrompt((type(id),press(enter))).

The Muddle Architecture

Muddle is shown as the evaluation component of Figure 3-2. The architecture of Muddle can described in two parts: a memory componet and a processing structure. First the structure of Muddle's memory will be discussed, followed by a description of the processing structure of the model.

Memory

Muddle uses an associative memory to store domain-specific and domain-general problem solving procedures. The model is consistent with assumptions made by other associative memory models. Specifically, Muddle's associative memory is patterned after the associative memory model of Raaijmakers and Shiffrin (1982; 1983). The Raaijmakers and Shiffrin model has been used to successfully model memory retrieval in list learning experiments.

Muddle's memory is a network of cues, images, and associations[1]. Cues and images are represented as propositions. Cues can be either goals or descriptions of the applications state. Each cue corresponds a condition of an individual production rule. Images consist of a list of goals and operators, corresponding to the action side of production rules. Associations represent the linkage between cues and images; the strength of an associtions reflects the frequency of successful associations between the cues and images. A cue is considered to be successfully associated with an image if the cue was a member of the cue set used to retrieve the image and if the actions of the image were all successfully executed.

Muddle's associative memory is searched using the sampling equation of Raaijmakers and Shiffrin. The sampling rule was chosen for two reasons. First, it has been used to account for a large number of list-learning experiments. Second, it produces a consensus model of memory retrieval (the sampling equation has been used by Anderson (1983) to model the search of associative memory through spreading activation). Other successful applications of the sampling equation include Walker and Kintsch's (1985) model of category member retrieval, and Kintsch and Mannes' (1987) model for generating scripts from memory.

Raaijmakers and Shiffrin's (1981) sampling equation is a form of Luce's (1959) choice axiom. It states that the likelihood of an item in memory being recalled is equal to the ratio of that item's strength to the sum of the strengths of all memory items.

$$\text{SamplingStrength}(I_i,C_1,C_2,\ldots,C_n) = \frac{\prod_{j=1}^{n} AS(I_i,C_j)}{\sum_{k=1}^{m}\prod_{j=1}^{n} AS(I_k,C_j)}$$

[1] A list of actions is called an image to be consistent with the terminology developed by Raaijmaker and Shiffrin (1980; 1981). In the context of the Muddle model, an image always refers to a list of actions.

Where the term *I* refers to images, C refers to cues, and $AS(I_i,C_j)$ refers to the associtive strength between image I_i and cue C_j.

An examination of the Raaijmakers and Shiffrin sampling equation shows that it produces strong inhibitory effects if cues are not associated with an image. If a cue and image are not associated then their associative strength is zero. When this value is used in the sampling equation, it results in the elimination of all images that are not associated with all the cues in the cue set. This inhibitory effect was too powerful: few images match the exact cues encoded in learning; retrieval failures would be legion. For this reason, Raaijmakers and Shiffrin assumed that all cues are connected to images by nonzero strengths. This was done by including a *residual* value between unassociated cues and images as a parameter in their model. Muddle also assumes that only nonzero connections exist and sets the residual value between unassociated cues and images to a small value. However, Muddle uses different residuals depending on whether the unassociated image is a general problem-solving method or a domain-specific method. The residual value is set higher for problem solving methods (0.09) than for domain specific methods (0.01). The small residual strength between an unassociated cue and image is an expression of uncertainty about the correctness of an unassociated image's actions in a particular context. The difference in residual values reflects an assumption that in uncertain situations general problem-solving methods should be preferred over domain-specific methods.

Muddle retrieves procedures from memory by probing with a set of cues (goals and system states). Associations between cues and images in memory determine what images are retrieved. Any images associated with the probe cues are considered as candidates for retrieval. The candidate images are ordered by their sampling strengths using the sampling equation; simulating the search of associative memory by an activation mechanism, with the image with the greatest sampling strength becoming the retrieved image.

In essence, Muddle implements a production system built using an associative network . Here the goals and conditions appearing on the production's condition side are used as cues to probe memory. The sampling equation performs the functions of the pattern matcher and conflict resolution mechanism. The association between cues and images determines which actions are candidate actions, while the sampling strength of each image is used to resolve conflicts when several images are retrieved.

The associative net representation has two benefits. One benefit is that partial matching is performed automatically by the model. Since goals and conditions are individually associated with actions, the condition side of the production is no longer an all-or-none bundle of conditions that must be matched. If several conditions associated with an action are absent, an action can still be retrieved and executed. Another benefit is that an associative representation incorporates a well-tested theory of forgetting. This representation conforms to the Raaijmakers and Shiffrin model of forgetting as retrieval failure. Raaijmakers and Shiffrin's model can account for interference effects. Indeed, this is how Muddle predicts which procedures will be forgotten and which will be remembered.

The Processing Structure of Muddle

Muddle cycles through a sequence of five stages: 1) An examination of the application's state; 2) Memory retrieval; 3) Solving for a retrieved method; 4) An evaluation of the solution; and 5) Generalization and storage of the episode in memory. These steps are represented diagrammatically in Figure 3-3.

In the first stage the model examines the description of the system and selects the parts of the description used to probe memory: The description of the system presented to Muddle are the functional stimuli while the cues selected by Muddle are the nominal stimuli. Unfortunately, little is known about what novice users attend to when they examine a menu. Muddle's selection of cues from system descriptions is a crude approximation to the cue selection processes users use. The assumption is that novices attend to unique objects appearing on the menu. If several members of a class of objects, such as ID-letters and ITEMS, are present on the menu, these objects will not be used as cues. Also, Muddle embodies the assumption that any relations between items (e.g., correspondences between ID-letters and ITEMS) on the menu must be computed and because of this, will not be considered in the selection process. Muddle focuses on unique descriptions of objects on the menu. For the spelling tasks menu shown in Figure 3-1, the model selects the menu title and the two system prompts as the salient cues.

Cue selection is further complicated by minor changes in the state of the system that take place. These changes can include: the changing of menu prompts; the highlighting of menu items; the presentation of system messages; and the display of character strings on the prompt line. When these minor changes of state occur, the model focuses only on the changes and ignores any unique objects present before the change of state. For instance, if the main prompt of the spelling-tasks menu changed, only the new main prompt would be considered as a cue. The menu title and secondary prompt would be ignored because they were present before the prompt changed. This process reflects the assumption that the salience of cues lessens with time.

In addition to the cues gleaned from the system-state representation, the last successful method performed by the model is also used as a cue. This cue provides the model with information about how methods are sequenced. These temporal cues form the sequential control-structure of the model. Temporal information has been used by Kintsch and Mannes (1987) to retrieve script events from an associative memory.

Once the model has selected a set of cues, the simulation begins the second processing step of searching memory for an associated image. The memory search process begins by discarding any cues in the probe set that have no associations. After the cue set has been culled, the remaining cues are used to activate associated images. Next, the cues are used to determine the strength of each of the images using the sampling equation. After the image strengths are computed, the image with the greatest strength is retrieved.

Following the retrieval of a image, the model attempts to execute the list of actions contained in the retrieved image. Each of the actions is executed in the order it appears on the list. These actions can be checks of application's state, operations that change the state of the

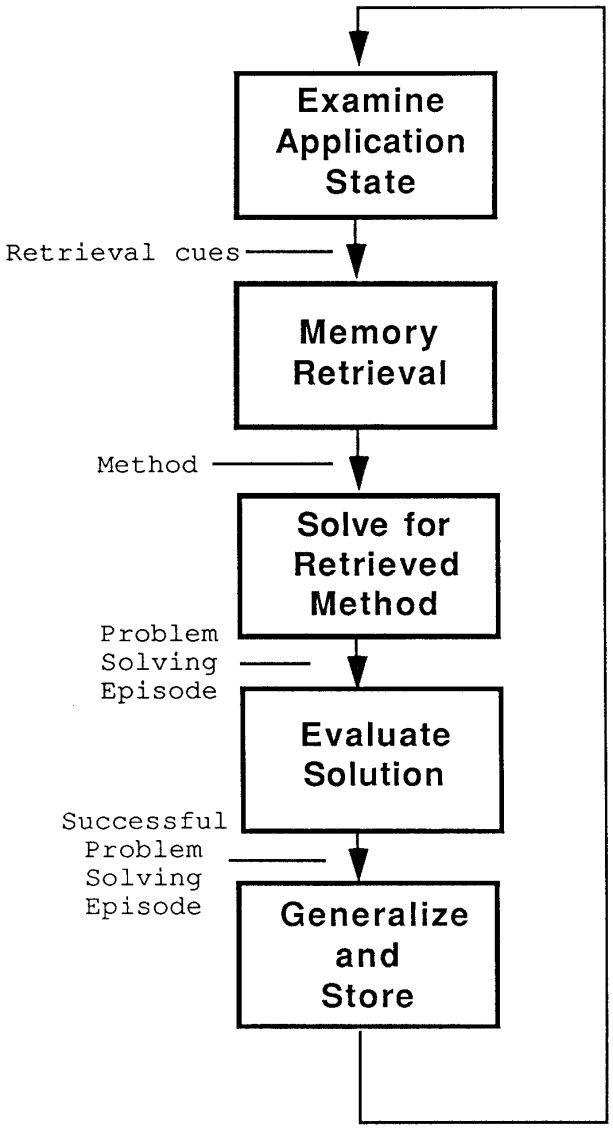

Figure 3-3. Diagram representing the processing structure of Muddle.

application, and subgoals spawned by the method. Once all the retrieved actions are executed, the system goes onto the next step of evaluating the results of its actions. However, if the retrieved actions do not lead to a physical action, the system backtracks and retrieves the image of next-highest strength. Backtracking continues until a image is retrieved which results in the execution of physical actions.

Muddle has a very simple mechanism for evaluating its actions: It relies on the system to tell it if its actions were correct. The simulation begins its evaluation by examining the state of the system after an operation is performed. If the system changed state, and no error message was generated, the simulation assumes its actions were successful and proceeds to the next step. However, if the system did not change state or a change of state indicates that an error occurred, the model assumes its actions were wrong and attempts the step of the problem again. If the model fails in its second attempt at a step, it is given the correct solution. This simulates the actions of a computer-aided instruction package used in the experiments. When given the solution, the model attempts to derive it for itself using problem solving methods. If the model is able to derive the solution it executes the solution and proceeds to the next step. However, if the model is unable to derive the given solution it simply executes the actions of the solution and proceeds to the next step. Derived solutions can be used by the model to generalize from the given solution.

In the fifth and final step, the simulation generalizes its representation of the current problem-solving episode and updates its memory. Each problem solving episode is represented by the set of cues used to probe memory and the successfully-executed retrieved image. The model makes generalizations by using semantic information present in the propositional representations of cues and images. In essence, generalizations are made by variablizing references to instances of objects described by the propositions in the cues and the image. For example, a reference to an ID-letter "a" can be generalized by variablizing the instance "a" so that it refers to any ID-letter. Once the simulation has generalized the cues and images from the problem-solving episode, memory is updated. The simulation updates memory by incrementing the strength of existing associations between cues and images, or by storing new cues, images, and associations in memory.

Determinants of Muddle's Behavior

Problem Solving

Like ACT* and SOAR, Muddle assumes that problem solving is the basic unit of thought and that learning occurs as the result of storing the results of problem solving episodes in memory. It also assumes that the encodings formed during learning determine retention. Specifically it assumes that cues (goals and states) used to retrieve general problem-solving methods can also be used to retrieve the domain-specific methods derived from successful application of problem-solving episodes. The model also assumes that a cue's ability to retrieve

an action is determined by the frequency with which it has been associated with the action. Clearly if these assumptions are correct the model should be able to successfully mimic how people learn.

Muddle's Initial problem solving knowledge is represented in Table 3-3 as production rules[2]. Variables are represented using capital letters in these rules (e.g., INSTRUCTIONS). In addition, the priority of rules is shown by their order of appearance. For example, Muddle will attempt to type a string (production P3) before checking whether that string refers to a category from which a member must be chosen (production P4).

The problem-solving rules in Table 3-3 provide Muddle with the ability to follow prompts and make correspondences between task descriptions and menu items. These rules make no reference to the menu tasks and an analysis of their system yields little task specific knowledge. Muddle encodes system-centered domain-specific knowledge because its problem-solving methods rely on system information only.

As the fifteen rules in Table 3-3 indicate, relatively little problem solving knowledge is needed to perform the menu tasks in this study. A learner needs to know how to follow instructions contained in the prompts, how to make correspondences between items referred to in prompts and menus, and how to make correspondences between task descriptions and menu items. The amount of knowledge needed to perform the tasks was reduced by carefully wording the task descriptions and by preventing the model from deviating from the solution path (a function performed by emulating the actions of the CAI package built into the system participants used in this study). The task descriptions were worded so that phrases appearing as menu items also appeared in the task descriptions. This greatly reduces the difficulties participants have in making menu selections since matches can be made based on identical wording between task descriptions and menu items.

The model assumes that inexperienced users employ a form of means-ends analysis in their search for solutions to a task. Means-ends analysis is a search method in which an action is selected based on its similarity to the goal state: The action whose result is most similar to the goal state is chosen. Participants in this study are given a description of the task they are to perform. This description is assumed to serve as the description of the goal state that participants use to guide their behavior. For example, when participants are forced to make a menu selection it is assumed that they will chose the menu item that most closely corresponds to the description of the task.

The model is given the knowledge necessary to convert the directions contained in the prompts into procedures. These methods are shown in Table 3-3 as prompt-following rules. These two rules translate the instructions contained in the prompts into goals to be solved. However, following the directions given by these prompts takes a good deal of problem solving. This consists of finding correspondences between the references made in the prompts and the menu. For example the instruction to type an ID-letter requires that the user find out what an ID-

[2] The initial problem-solving knowledge is represented as product rules because this representation is easier to read than the associative memory representation. The associative memory representation of this knowledge is given in Appendix B.

Table 3-3.

Production Rule Representation of the Problem Solving Knowledge With Which the Muddle Model is Initialized

---

Prompt Following Rules

P1 **IF** (mainPrompt(INSTRUCTIONS))
THEN (INSTRUCTIONS)

P2 **IF** (secondaryPrompt(INSTRUCTIONS))
THEN (INSTRUCTIONS)

Rules For Typing Character Strings and Pressing Keys

P3 **IF** (type(CATEGORY(INSTANCE)))
THEN (choose(CATEGORY(INSTANCE)), typeString(INSTANCE))

P4 **IF** (type(INSTANCE))
THEN (instance(INSTANCE),typeString(INSTANCE))

P5 **IF** (press(INSTANCE))
THEN (isaKey(INSTANCE),pressKey(INSTANCE))

Rules for Choosing Things

P6 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(GOAL)),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(GOAL)))

P7 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(A(B(C(D))))),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(A(B(C)))))

P8 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(A(B(C)))),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(A(B))))

P9 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(A(B(C(D))))),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(A(B(C(E))))))

P10 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(A(B(C)))),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(A(B(D)))))

P11 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(goal(A(B))),corresponds(CATEGORY(INSTANCE),
OTHERCATEGORY(A(C))))

P12 **IF** (choose(CATEGORY(INSTANCE)))
THEN (corresponds(CATEGORY(INSTANCE),OTHERCATEGORY(A)),
corresponds(OTHERCATEGORY(A),SLOT(empty)))

P13 **IF** (choose(CATEGORY(INSTANCE)))
THEN (task(CATEGORY(INSTANCE)))

Rules for Following Instructions that Suggest Two Possible Actions

P14 **IF** (do(or(A,B)))
THEN (A)

P15 **IF** (do(or(A,B)))
THEN (B)

---

letter is by locating a reference to it on the menu. The problem solving methods are represented as the "rules for choosing things" in Table 3-3. These rules make up the largest category of problem solving rules given to the model. The major reason is that the first six rules represent ordered preferences for partial matches between task descriptions and descriptions of the menu. The seventh rule of this set states that if there is an empty slot then it should filled in. The last rule of the set is essentially a mechanism for guessing. It states that if you are asked to provide a category instance (an INSTANCE of a CATEGORY) and you can find an instance, use it.

The seven rules for choosing things given in Table 3-3 only approximate the strategies people employ when choosing menu items. These rules restrict the ability of the model to emulate the behavior of users in two ways. First, the rules for choosing things cannot be used to recognize synonyms. The result is that the model is unable to recognize that a task description (e.g., erase document) corresponds to a menu item that contains a synonym (e.g., delete document). The second limitation of these rules is that they do not recognize when a description for a set of tasks is given in place of descriptions of specific tasks. This means that the model cannot recognize that the menu item "typing tasks" refers to the task of editing a document.

The problems caused by the limitations of the model's methods for choosing things are circumvented by adding to the task descriptions provided to the model. The problem of identifying synonyms is circumvented by using the terms from the menus in the task descriptions. The problem of identifying general category descriptions is circumvented by adding the general category descriptions to the tasks descriptions provided to the model. While these solutions allow the model to evade problems making menu choices they also limit the model's ability to emulate users making menu choices. The Muddle model is unable to predict when people will, correctly or incorrectly, decide that a menu item is synonym for the task they are performing or that a general description given for a set of tasks corresponding to the particular task being performed.

These rules for choosing things can be seen as a form of the identity heuristic of Lewis (1986; 1987). Lewis uses the identity heuristic to make inferences about the connection between observed events. The identity heuristic rests on the assumption "Separate occurrences of a thing are not coincidental" (Lewis, 1985). The identity heuristic is used in this study to infer the correspondences between referents in instructions and menus, and to match task descriptions with menu choices.

The rules for typing character strings and pressing keys decide if an operation should be performed (typeString, or pressKey). The rules for following instructions that suggest two possible actions are special case rules. The menu system used in this study occasionally directs users to perform one of two possible actions. These two rules are used to decide which of the two actions is to be performed.

There will be times when the instructions, heuristics, or previous experience will mislead the model and result in an error. If the model were not told that it had made an error it would go blindly on searching for a solution. The model, like people in the experiment, is stopped when

an errors occurs and is told the correct action on the third error. When Muddle is told the correct answer it merely stores the correct answer and does not interpret why its previous actions were wrong. This is a good description of how participants in previous studies responded to errors when using the menu simulation.

If the above restraints were not imposed, Muddle would continue on after an error had occurred. This behavior is similar to that shown by people learning to use computers (Lewis and Mack, 1982). The model, however, differs from human behavior since it does not try to construct an explanation for why its behavior is correct.

In summary, there are two ways in which the model can solve a problem. First the model can apply weak problem-solving methods to a domain. This yields some knowledge of the domain: specifically, what method worked and what methods did not work. This knowledge is stored in memory, allowing the model to solve efficiently the same problem again.

The second way the model can solve problems is by applying domain-specific methods to the problem. Images that are retrieved using the cues from the current context can be used to guide behavior. These extrapolations eliminate the amount of search the model must perform. When successful, extrapolations reduce the amount of search required to solve a problem

Learning Mechanisms

Muddle assumes that learning occurs as the result of an analysis of problem solving episodes. When the model successfully completes a step of the menu tasks it generalizes the cues used to probe memory and the problem solving method and stores them in memory. The cues are stored as individual associations with the successful image.

The model can transfer knowledge in three ways. The first way is to generalize is representation of problem solving episodes by variablizing instance information contained in cues and images. This increases the scope of the rules, allowing them to match a greater number of system states. The second way a transfer can occur is through partial matching. Since the model stores its procedural knowledge in an associative memory, its procedures can be retrieved using a subset of the cues associated with an image. This can happen when cues used in one context are missing from a later context. The third way the model can transfer knowledge is through the associative strengths between cues and images. These associative strengths determine which method will be applied to a particular context, this can result a method being applied to a new context.

An Example of Muddle Learning to Perform a Menu Task

To see how Muddle's mechanisms work together to explain the acquisition and transfer in menu tasks we will trace Muddle's behavior as it learns to perform the steps needed to complete the check spelling task described in Table 3-1. The five steps making up this task are given in Table 3-4. At the first step, Muddle is presented with a description of the check-spelling task, the task-selection menu, and the knowledge that it has just obtained from the task description. First, Muddle selects a salient set of system cues. At this step these cues are the

menu's title and prompt. These cues are used to search memory. Since the only knowledge Muddle model has available is its general problem-solving knowledge (see Table 3-3), the only image retrieved from memory is one that interprets the prompts and spawns the subgoals to type an ID letter, and to press the ENTER key.

Muddle tries to solve for the goal of typing an ID-letter by using this goal to probe memory. This goal retrieves a method to choose an instance of an ID-letter and then typing the string corresponding to that ID-letter. Muddle then tries to choose an ID-letter. Muddle has several methods for making correspondences between menu items and task goals. It eventually finds one that succeeds, selecting the ID-letter "d". Muddle then generalizes and stores this method in memory producing a general method for selecting ID-letters. After storing this method, Muddle turns to the next goal which is typing the string "d". It does this and stores a general method for typing an ID-letter, which is to first choose an ID-letter and then type it. Muddle then satisfies the next goal by pressing the ENTER key. After the ENTER key is pressed, the menu system changes state, presenting the spelling-tasks menu. From these changes Muddle concludes it performed the correct action and stores the method "type ID, press ENTER" with the cues used to retrieve the problem-solving method that generated this solution. This change is reflected in the propositional representation of the state of the system.

Table 3-4.
Steps of the Check-Spelling Task.

Step	Action	Menu	Prompt
1	d ENTER	Task Selection	Type ID letter to choose ITEM; press ENTER
2	a ENTER	Spelling Tasks	Type ID letter to choose ITEM; press ENTER
3	intro ENTER	Spelling Tasks	Type document name; press ENTER
4	paper ENTER	Spelling Tasks	Type diskette name; press ENTER
5	ENTER	Spelling Tasks	Type ID letter to choose ITEM; press ENTER

Muddle is now ready to perform the second step of the task. It gathers a new set of cues by noting what salient changes took place in the representation of the system state. It combines these cues with a description of the last goal solved for (type ID, press ENTER) and probes memory. This time the cues retrieve the domain specific associations that were added on the previous step. Muddle uses these associations to perform this step correctly. The successful use of these associations increases the associative strength between the cues present and their associated images.

The next two task steps require Muddle to enter document and diskette names. Since both of these steps are novel, the general problem-solving methods are used to find solutions in same manner as the first step of this task. The performance of these two steps results in the

addition to memory of domain-specific methods for choosing and entering document and diskette names.

On the fifth and final step of the task Muddle is again faced with a context in which an ID-letter must be entered. Muddle retrieves the domain-specific method for selecting an ID-letter created in the first step and applies it to this new context. The result is that Muddle selects ID-letter "a" because it is the only ID-letter that corresponds to a task goal (check document). However, this is not the correct action and Muddle is told it failed by the CAI package. It attempts this step of the task again and repeats its selection of the ID-letter "a", and makes a second error. After this Muddle is given the correct solution: press the ENTER key. Given this solution and its existing knowledge Muddle attempts to derive this answer. It is unable do this, however, since pressing the ENTER key does not have any correspondence to the goals given in the task description. Because Muddle is unable to derive the answer, it simply performs the action and stores the given solution in memory. The performance of the model leads to the prediction that participants will form new associations for entering ID-letters, document, and diskette names, and will learn how to leave the menu. The model also predicts that participants will make an error on the last step of the task by typing ID-letter "a".

Predictors of Performance

There are four phenomena that the model attempts to explain: training time, training errors, retraining time, and retraining errors. Training time is the time required for participants to complete a task or tasks steps during the training session. Training errors are deviations from the solution path made by participants during their attempts to perform the training tasks. Retraining time is the time required for participants to perform tasks during the retraining session. Retraining errors are the errors made by participants while performing the retraining tasks.

The time spent leaning to perform a task should increase with the amount of time spent problem solving. Tasks that are difficult to learn are those that require extensive search before a solution is found, or result in people trying numerous actions before finding the correct one. In either case, given an understanding of the problem solving methods people have at their disposal, and when they will chose one method over another, we should be able to predict the difficulties they will have during training. The predictors of the model reflect how much processing the model had to do into order to perform a task.

Errors can be divided into three types: 1) inappropriately applied problem solving methods; 2) extrapolations from previous experiences that produce incorrect actions; and 3) slips which result in typographic errors. While slips are difficult to predict, misapplication of problem solving methods and extrapolations from experience can be predicted.

The predictors in the model are: 1) the number of new associations added to memory, 2) the number of new cues added to memory; 3) number of new images added to memory; 4) errors made by the model; and 5) number of processing cycles.

Summary

This chapter presented a description of the Muddle model. The purpose of this model is to account for the initial learning and retention of the procedures needed to perform tasks using a menu-based word processor. Muddle assumes that novice users of computer applications learn to perform tasks through the system of general problem-solving methods to the information provided by the application. In addition, Muddle assumes that what users encode from these experiences is system-centered knowledge: They learn what *type* of actions to take in particular systems contexts rather than what actions to take to accomplish a task goal.

Muddle assumes that procedural knowledge can be represented in a associative network. The associative network allows Muddle to employ an interference model of forgetting and allows it to perform partial matching.

In the chapters that follow the performance of Muddle is compared with the behavior of novice users who are learning and remembering how to perform utility tasks using a menu-based word processor. Muddle's predictors, new associations and errors, are used to account for the observed performance of participant's performing menu tasks.

CHAPTER IV
EXPERIMENT 1

Purpose

The present chapter has two goals. The main goal of this chapter is to see if the Muddle model can account for the acquisition and transfer of a cognitive skill. This chapter uses data from an expanded experiment first reported by Polson, Muncher, and Engelbeck (1986). This experiment was originally designed to provide a rigorous test of the transfer assumptions of Cognitive Complexity Theory (Kieras and Polson, 1985). The data from this experiment will also be used to test the transfer assumptions underlying the Muddle model.

The second goal of the chapter is to compare Muddle's account of acquisition and transfer with that of Cognitive Complexity Theory. Since a good explanation of these data has been provided by CCT, this experiment provides a good opportunity to compare directly Muddle's predictions with those of CCT.

This chapter begins with a description of the experiment itself. Next, a brief description of CCT is given and the derivation of its predictor of performance is described. After this discussion, the derivation of Muddle's predictors is described. Finally in the results and discussion section, the Muddle model's ability to account for the results of the experiment is discussed and its account is compared to the account given by CCT.

Method

Participants

The results reported below are from two replications of the experiment first reported by Polson, Muncher, and Engelbeck (1986). One hundred and seventy-five students participated in this experiment. Participants were recruited from introductory psychology courses at the University of Colorado and participated in exchange for credit toward fulfillment of a course requirement. Fifteen participants were run in each of five experimental groups in the first replication reported in Polson, Muncher, and Engelbeck (1986), while twenty participants were run in each group in the second replication. This resulted a total of 35 participants in each of the five experimental groups.

Hardware and Software

The experimental tasks were performed using a simulation of an IBM Displaywriter menu system incorporating a simple computer assisted instruction package (CAI). Both the simulation and CAI package were written in Fortran 77 and ran on a DEC 11/780. This system and the basic

training procedures used are same as those used by Polson, Kieras, Engelbeck, and Willer (1984), Polson and Kieras (1984), and Engelbeck (1986).

Menu Tasks

Three types of menu tasks were used in this study: *Parameter-entry* tasks, *forms-fill* tasks, and *prompt-answering* tasks. The *parameter-entry* tasks required a series of menu traversals until a parameter-entry menu was reached. At this final menu, participants had to change the value of one of several parameters (e.g., the line spacing for a document from single to double spaced) and then leave the menu. Completion of *form-fill* tasks also required a series of menu traversals until a form-fill menu was reached. At this final menu, participants had to fill in the empty slots of the menu and then leave the menu. The *prompt-answering* tasks differed from the *parameter-entry* and *form-fill* tasks. To perform these tasks, participants traversed a series of menus, finally reaching a menu that prompted them for information needed by the system. The system generally prompted participants for document, diskette, or supplement names. The prompt-answering tasks never involved the use or the display of parameter-entry or form-fill menus.

In addition to making menu choices, changing parameters, filling slots, and answering prompts, participants in this study had to manipulate diskettes. Several tasks also involved the manipulation of diskettes using a simulated diskette unit. The physical diskette unit was simulated with a diagram of the diskette unit presented on a CRT screen. The simulated diskette unit allowed users to load and unload diskettes by pressing labeled function keys and to insert diskettes by answering prompts.

Nine utility task were selected for use in this experiment and are listed in Table 4-1. The first four tasks listed in Table 4-1 were *parameter-entry* tasks. Tasks 1 and 2 were tasks which required the changing of default format parameters. These task shared the same initial sequence of steps. Tasks 3 and 4 were also *parameter-entry* tasks, but these tasks required the loading of a diskette in the right slot of the diskette unit and the changing of format parameters specific to a document. Like tasks 1 and 2, tasks 3 and 4 shared the same initial sequence of steps. Tasks 1 and 3 were line printing tasks and required the changing of a parameter that specified on which line the first typing line of a document would print. Similarly, tasks 2 and 4 were line spacing tasks and required the changing of parameters specifying the line spacing for a document. The two line printing tasks and the two line spacing tasks shared a common terminal menu and hence, shared the same sequence of terminal steps.

Tasks 5 and 6 were diskette duplication tasks. These tasks required the loading of a diskette into the right slot of the diskette unit. After the diskette was loaded, these tasks required a sequence of menu transitions until a form-fill menu was reached. On reaching this menu, participants had to fill in the empty slots of the menu. After filling in the slots on the menu, participants had to remove a program diskette from the left slot of the diskette unit and load a

second work diskette in its place. Participants then had to leave the diskette unit and then successfully leave the form-fill menu.

Task 7 was a *prompt-answering* task, while tasks 8 and 9 were *form-fill* tasks. These three tasks did not require any manipulation of diskettes and were included to evaluate nonspecific transfer effects with a control group. The control group learned to perform these three tasks and then the four *parameter-entry* tasks (tasks 1, 2, 3, and 4). If nonspecific transfer is a large component of the transfer process then the four *parameter-entry* tasks should be less difficult thanpredicted by either model. It should be pointed out that these *parameter-entry* tasks do share common methods with the first three tasks and, consequently, transfer is predicted by each model.

Table 4-1.
Description of the Menu Tasks for Experiment 1

Task Number	Task Description
Default Format Tasks	
1	Change the default first typing line from 5 to 7.
2	Change the default spacing from double to single.
Document Format Tasks	
3	Load diskette. Change the first typing line for a document from 5 to 7.
4	Load diskette. Change the spacing for a document from double to single.
Work Diskette Duplication Tasks	
5	Load diskette. Duplicate the diskette.
6	Load diskette. Condense the diskette.
Miscellaneous Tasks	
7	Print a document.
8	Check a document for spelling errors.
9	Change a document name.

Design

The main variable manipulated in this study was the presentation order of highly similar menu task pairs. Table 4-2 shows the presentation order of the tasks for the four experimental groups and the nonspecific transfer group described above. The presentation orders for the four experimental groups were generated by the factorial combination of the location in the training sequence of the duplicate diskette tasks and the two possible orders on the two pairs of *parameter-entry* tasks. The two members of each task pair always appeared in the same order.

Table 4-2.
Task Training Orders for the Groups in Experiment 1

Group	Task Number						
	Serial Position						
	1	2	3	4	5	6	7
1	1	2	3	4	5	6	
2	3	4	1	2	5	6	
3	5	6	1	2	3	4	
4	5	6	3	4	1	2	
C	7	8	9	1	2	3	4

Procedure

Each task was learned to a criterion of three consecutive correct trials using a serial anticipation method. The menu simulation presented participants with the appropriate system display for each step in a task. Participants had to anticipate the correct action for each step. If a correct action was made, the simulation responded as the actual menu system would. However, if a wrong action was performed the CAI package intervened. The CAI package provided minimal feedback on a first error. The style of this feedback was consistent with the actual word processor. On the second error, the CAI package assumed control and presented participants with detailed feedback. The simulated menu display was replaced with a feedback screen which provided participants with information about the current step, the goal for the step, and the correct user action for the step. Participants were allowed to study feedback screens for as long as they wished. After finishing with a feedback screen, the menu display for the current step was redisplayed, and the participant had to give the correct action.

The manipulation of presentation order resulted in a number of groups whose learning curves where not the characteristic monotonically decreasing curve found in most learning studies. An adequate theory of skill acquisition and transfer must be able to account for these observed differences in learning curves. It was for this reason that these data were used to test the transfer assumptions of CCT. And, it is for this reason that these data are used to test the transfer assumptions of the Muddle model.

Testing the Learning and Transfer Assumptions of the Models

Cognitive Complexity Theory

First we turn to Cognitive Complexity Theory and how it explains the data from this experiment. This section begins with a short overview of Cognitive Complexity Theory and then describes how it accounts for the acquisition and transfer of the cognitive skills needed to perform the menu tasks used in this study.

Kieras and Polson (1985) proposed a quantitative theory of human interaction based on the GOMS model of Card, Moran, and Newell (1980; 1983) and Anderson's (1983) theory of cognitive skills embodied in ACT*. While Kieras and Polson assume the content and structure of the knowledge needed to perform tasks is correctly described by the four categories of the GOMS model, CCT differs from the GOMS model by representing procedural knowledge as productions. Kieras and Polson choose the production system representation for several reasons. First, production rules have been argued to be cognitive units. Specifically, Anderson (1983; 1987a) argues that productions are the basic computational units of cognition. The assumption that productions are the basic cognitive units permits CCT to make quantitative predictions of performance. Second, production system representations provide a convenient uniform representation for procedural knowledge. This makes the tasks of modeling acquisition and transfer easier.

The GOMS model representation for each of the menu tasks used in this experiment has a very simple structure. Each task is represented as a linear sequence of steps. However, each step is complex and requires the execution of a method. The methods common to all tasks are: load a diskette containing documents into the right slot of the diskette drive, go to the next menu, select an item from a menu, enter a parameter into a menu, enter a diskette name, and enter a document name. Figure 4-1 shows the GOMS model for the check spelling task described in Table 3-4.

The production system representation of this task has a structure identical to that of the GOMS model. Each rule in the production system model executes a single step, calling a method if necessary. The methods perform the operations necessary to complete each step. For example, the method to enter a document name reads the document name from the instructions, enters the name in response to the appropriate prompt, verifies the entry, and returns to the calling routine.

Cognitive Complexity Theory incorporates a transfer model that is a modern formulation of a common elements theory of transfer proposed by Thondike and Woodworth(1901). Thorndike and Woodworth stated that the performance of one task facilitated the performance of a second task only to the degree that they shared "common elements." However, Thorndike and Woodworth never gave a clear statement of what were the common elements. CCT accepts Thorndike and Woodworths' characterization of transfer and assumes that the common elements shared by tasks are productions (Kieras and Bovair, 1986). Kieras and Polson assume that productions are cognitive units and that training time is a function of the number of new productions that must be acquired to perform a task. Indeed, Kieras and Polson offer the following equation to explain the

observed patterns in training times:

$$\text{Training time} = N * T + C.$$

Where N is the number of rules that must be learned, T is the training time required to learn each new rule, and C is a constant representing the time needed to make the criterion task run and other parts of the training procedure that do not involve learning processes (Polson, 1988; Kieras, Bovair, and Polson, 1988).

Goal: CHECK-SPELLING
Goal: GOTO-CHECK-SPELLING-MENU
[USE GOTO-NEXT-MENU-METHOD]
Goal: VERIFY-GOTO-CHECK-SPELLING-MENU
Goal: SELECT-CHECK-SPELLING-FROM-MENU
[USE SELECT-ITEM-FROM-MENU-METHOD]
Goal: ENTER-DISKETTE-NAME
[USE ENTER-DISKETTE-NAME-METHOD]
Goal: ENTER-DOCUMENT-NAME
[USE ENTER-DOCUMENT-NAME-METHOD]
Goal: VERIFY-SPELLING-CHECK-COMPLETE
Goal: GOTO-TASK-SELECTION-MENU
[USE GOTO-NEXT-MENU-METHOD]

Goal: VERIFY-GOTO-TASK-SELECTION-MENU

Figure 4-1. A GOMS analysis of the check spelling task.

This experiment tested two transfer predictions made by CCT. The first was that once a common method was acquired it transfers perfectly to any task later in the training sequence making use of the method. The second prediction involved shared steps in the pairs of tasks described in Table 4-1. CCT assumes that transfer occurs at the level of individual shared rules. Thus, common initial and final sequences of steps are assumed to transfer perfectly.

The magnitudes of transfer effects were calculated from the number of common rules in the representations of common methods and identical sequences of steps. The mean time to the learning criterion for a task was determined by counting the number of *new rules* a user must learn to master the task. If a task appears in the first serial position of a training order, then all rules are new and the participant must acquire all of the rules representing the individual steps of the tasks as well as the common methods used in the task.

Tables 4-3 and 4-4 present the calculations of the number of new rules for the first experimental group and the control group respectively. The rows of these tables show the components making up the tasks while the columns show each task in its serial position. There are three possible entries in the tables: an asterisk; a zero; or a number greater than zero. An asterisk means that the component listed in the row did not occur for that task. A value of zero means that this component occurred in the task but that the rules needed to perform this task component had been acquiredon a previous task. Finally, a value greater than zero is the number of new rules that must be acquired to perform the task component. The sum of new rules is given at the bottom of each table. This sum is the total number of new rules needed by the model to perform each task in its given serial position.

Table 4-3.
Cognitive Complexity Theory's Calculation of the Number of New Rules for Each Position in the Training Sequence for Experimental Group 1

	Number of New Rules					
Subsequence or	Task Number					
Common Method	1	2	3	4	5	6
Task						
Steps Unique to Task	1	1	1	1	5	5
Initial Steps for Default	3	0	*	*	*	*
Initial Steps for Document	*	*	8	0	*	*
Final Steps for Typing Line	7	*	0	*	*	*
Final Steps for Spacing	*	7	*	0	*	*
Steps for Insertion of Diskette	*	*	*	*	20	*
Common Methods						
Load Diskette	*	*	12	0	0	0
Go To Next Menu	5	0	0	0	0	0
Select Item from Menu	5	0	0	0	0	0
Enter Parameter	5	0	0	0	*	*
Enter Diskette Name	*	*	5	0	0	0
Enter Document Name	*	*	5	0	*	*
Number of New Rules	26	8	31	1	25	5
Total Number of Rules	26	26	53	53	52	52

Table 4-4.

Cognitive Complexity Theory's Calculation of the Number of New Rules for Each Position in the Training Sequence for the Control Group

Subsequence or Common Method	Number of New Rules: Task Number 7	8	9	1	2	3	4
Task							
Steps Unique to Task	9	9	11	1	1	1	1
Initial Steps for Default	*	*	*	3	0	*	*
Initial Steps for Document	*	*	*	*	*	8	0
Final Steps for Typing Line	*	*	*	7	*	0	*
Final Steps for Spacing	*	*	*	*	7	*	0
Common Methods							
Load Diskette	*	*	*	*	*	12	0
Go To Next Menu	5	0	0	0	0	0	0
Select Item from Menu	*	5	0	0	0	0	0
Enter Parameter	*	*	5	0	0	0	0
Enter Diskette Name	5	0	0	*	*	5	0
Enter Document Name	5	0	0	*	*	5	0
Number of New Rules	24	13	16	11	8	21	1
Total Number of Rules	24	28	36	26	26	53	53

Table 4-5 presents the sum of new rules needed for each task by their serial positions for the four experimental groups and the control group. CCT predicts that these corresponding training means will be a linear function of these entries. Table 4-5 shows that CCT predicts that the total training time for all six tasks for the four experimental groups should be constant and independent of training order; the same *unique* productions must be mastered independent of training order. This prediction is reflected in the sum of the number of new rules learned across serial positions.

CCT predicts the same number of productions must be learned by the four experimental groups. CCT predicts that rules representing common methods and common initial and final sequences of steps are transferred independent of the particular task context in which they were learned. However, CCT does predict large differences in serial position curves across groups. The number of *new* productions that must be learned to master a task is determined by position in training order and by the particular tasks that were learned previously.

The Muddle Model

We now turn to the Muddle model and how it explains the data from this experiment. This section begins with a short overview of the model and then describes its account for the acquisition and transfer of the cognitive skills needed to perform the menu tasks.

As was explained in the previous chapter, Muddle is a model of human computer interaction based on a problem-space architecture. The Muddle model embodies the assumption that the content and structure of the problem solving methods, in combination with its learning mechanisms, determine the structure of acquired knowledge. Muddle's problem solving knowledge serves two purposes. First, it gets behavior in a domain started and second, it provides successful problem solving episodes as inputs to the learning mechanism. Not surprisingly, the purpose of Muddle's learning mechanisms is to improve performance by reducing the amount of search needed to perform tasks.

The Muddle model has three learning mechanisms. The first is a caching mechanism which stores the results of problem-solving episodes in its memory as associations between goals and system states and operators and subgoals. Muddle's second learning mechanism is a generalization mechanism. This mechanism uses semantic information provided by the system to generalize over instance information. For instance, when typing an ID-letter "a", Muddle is able to examine the state of the system and infer "a" is a single instance of the category ID-letters. It is possible for the model to do this since the menu lists other instances of ID-letters ("b", "c","d", ...). The third mechanism used by Muddle to reduce search is associative strength. Since Muddle encodes its procedural knowledge in an associative memory and because associative strength

Table 4-5.
Task Training Orders for the Groups in Experiment 1 and the Number of New Rules Predicted by Cognitive Complexity Theory

Group	Task Number (Number of New Rules)							
	Serial Position							
	1	2	3	4	5	6	7	SUM
1	1(26)	2(8)	3(31)	4(1)	5(25)	6(5)		(96)
2	3(53)	4(8)	1(4)	2(1)	5(25)	6(5)		(96)
3	5(52)	6(5)	1(16)	2(8)	3(14)	4(1)		(96)
4	5(52)	6(5)	3(26)	4(8)	1(4)	2(1)		(96)
C	7(24)	8(13)	9(16)	1(11)	2(8)	3(31)	4(1)	(104)

determines likelihood of retrieval from this memory, the associative strengths between cues and images influences the order in which methods are applied to a problem.Just like CCT, Muddle incorporates a transfer theory that is a modern formulation of Thorndike and Woodworth's (1901) common elements theory of transfer. Where the common elements in CCT are productions, Muddle's common elements are associations between cues and images. However, this is not the only difference between CCT and Muddle. Since Muddle learns by applying problem solving and domain-specific methods to a problem, it is capable of making errors. Hence, Muddle predicts both positive and negative transfer. Positive transfer occurs when a problem-solving or domain-specific method is appropriately applied to a problem, and negative transfer occurs when a method is inappropriately applied to a problem and leads to an error. The result is a model of acquisition and transfer that assumes that training time is a function of the number of *new associations* needed to perform a task and the number of errors made by the model. Expressing the predictors of Muddle as an equation gives:

$$\text{Training time} = N * T1 + E* T2 + C.$$

Where N is the number of new associations that must be learned, T1 is the training time for each association that must be learned, E is the number of errors made, T2 is the time taken to recover from each error, and C is a constant representing the time needed to make the criterion task run and other parts of the training procedure that do not involve learning processes.

Muddle predicts that the total training time for all six tasks is a function of the associations that must be added to memory and what errors are made. Like CCT, Muddle predicts large differences in serial position curves across groups. The number of new associations added to memory is determined by when problem solving methods must be used or by encoding new cues.

The Muddle model predicts the magnitude of transfer effects in a manner similar to CCT. Transfer effects are calculated by counting the number of new associations the model adds to memory. The mean time to the learning criterion for a task is assumed to be determined by the number of *new associations* a user must learn to master the task and number of *errors* made during training.

Table 4-6 gives the number of new new associations and predicted number of errors for all five groups for each serial position. Muddle, like CCT, predicts that these corresponding training time means will be a linear function of the entries. Note, however, that sum of associations added and errors made given in Table 4-6 predicts that context effects should occur. This contrasts with CCT which predicts that no context effects should occur. Muddle's prediction of context effects can be traced to differences in associative strength. Differences in associative strength occur because the presentation order of tasks affects the pattern in which associations accrue strength. This produces different likelihoods for applying methods during training. The result is that several task orders led to the model's inappropriately transferring its knowledge in some situations but not in others. When this happened in the context of the simulation an error message was displayed that became a cue used to recall the appropriate method once it was acquired. This cue is stored as an association to the appropriate method. The result was that task orders that led to more errors

generally added more associations to memory.

Another difference is apparent from a comparison of Tables 4-5 and 4-6. CCT predicts that the control group should take the longest to perform the tasks of any group, while Muddle predicts it should take the least amount of time. These differences in predictions reflect Muddle's greater aggressiveness in transferring knowledge.

Despite these differences between the models, the predictions of the models are very similar. Indeed, the correlation between CCT's prediction of new rules and Muddle's prediction of new associations is .94. Given this high correlation and the knowledge that CCT gave a good account for these data, it is expected that Muddle should be able to give a good account for these data.

Table 4-6.
Task Training Orders for the Groups in Experiment 1 and the Number of New Rules Predicted and Errors by the Muddle Model

Group	Task Number (Number of New Associations, Number of Errors)							
	Serial Position							
	1	2	3	4	5	6	7	SUM
1	1(17,1)	2(0,0)	3(27,3)	4(0,1)	5(36,5)	6(12,4)		(92,14)
2	3(45,5)	4(0,0)	1(2,1)	2(0,0)	5(35,5)	6(12,4)		(94,15)
3	5(67,8)	6(10,5)	1(9,2)	2(1,1)	3(7,2)	4(0,1)		(94,19)
4	5(67,8)	6(10,5)	3(14,2)	4(0,0)	1(2,1)	2(0,0)		(93,16)
C	7(27,3)	8(8,2)	9(10,1)	1(7,1)	2(0,0)	3(18,4)	4(0,2)	(70,13)

Comparison of CCT and Muddle

Muddle and CCT differ architecturally and in the assumptions they make about how people encode and generalize procedures. The important difference between the models is their assumptions about how users structure their procedural knowledge. Both CCT and Muddle represent goal and system contexts in their representations of procedures. CCT relies on a strict representation of goals to control its behavior. Muddle, on the other hand, relies on system context and a weak encoding of task knowledge to control its behavior: Muddle links the selection

of methods to system states and knowledge of the last action. Indeed, Muddle is incapable of producing the hierarchy of goals assumed by the GOMS representation.

Results and Discussion

The dependent variable measured in this study was time to criterion. Time to criterion is the amount of time it takes a participant to complete a task, starting from their initial reading of the task instructions to their last action on their last correct trial of the task.

Both models were evaluated by fitting their predictors to time to criterion cell means for each task: a total of 31 cell means. The observed and predicted mean time to criterion are shown as a function of serial position for the five experimental groups in Figure 4-2.

CCT assumed that the cell means would be a linear function of the number of new rules; the slope of the best fitting straight line is an estimate of the training time per new rule. The intercept is the time taken in performing the criterion run and other processes not involving acquisition of rules. Recall the predicted number of new rules for each cell is shown in Table 4-5. Table 4-7 summarizes the fit of new rules to mean time to criterion. Table 4-7 shows that the observed training time per rule was 26.9 seconds, and the intercept was 84.1 seconds. CCT accounted for 88% of the variance of the 31 means. Two significant differences between observed and predicted values for the 31 means occurred. The first occurred in Serial Position 1 for Experimental Group 4; the other occurred in Serial Position 6 for the control group.

Like CCT, Muddle assumed that the cell means would be a linear function of number of new associations. The slope of the best fitting straight line is an estimate of the training time per new association. Table 4-8 shows the fit of the number of new associations fit to the 31 cell means. The observed training time per association was 21.1 seconds with an intercept of 177.7 seconds. This fit accounted for 94% of the variance of the 31 means. No significant differences were found between the observed and predicted values for the 31 means.

Additional analyses were performed using the individual times to criterion of participants. These analyses fit the predictors of CCT and Muddle to the 1078 individual times to criterion. These analyses tested the ability of the models to give a consistent account for both the individual task completion times and the mean times to criterion. Additionally, these analyses could be used to make comparisons between the two models.

The predictors of the models and each participants individual mean time to criterion were regressed against the dependent variable. Each participant's mean time to criterion was included in the analysis to account for individual differences (Pedhauzer, 1982).

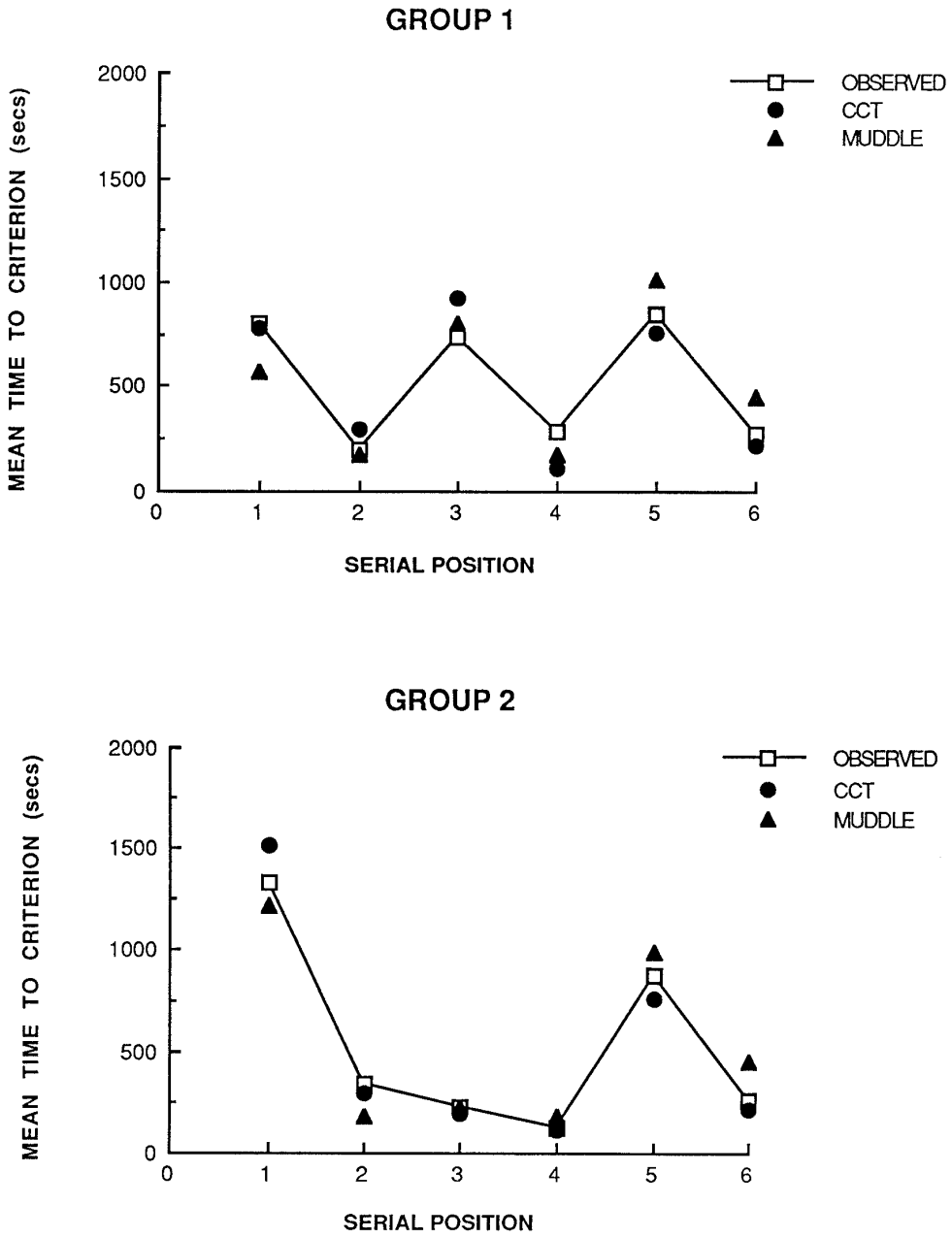

Figure 4-2. Observed and predicted mean times to criterion.

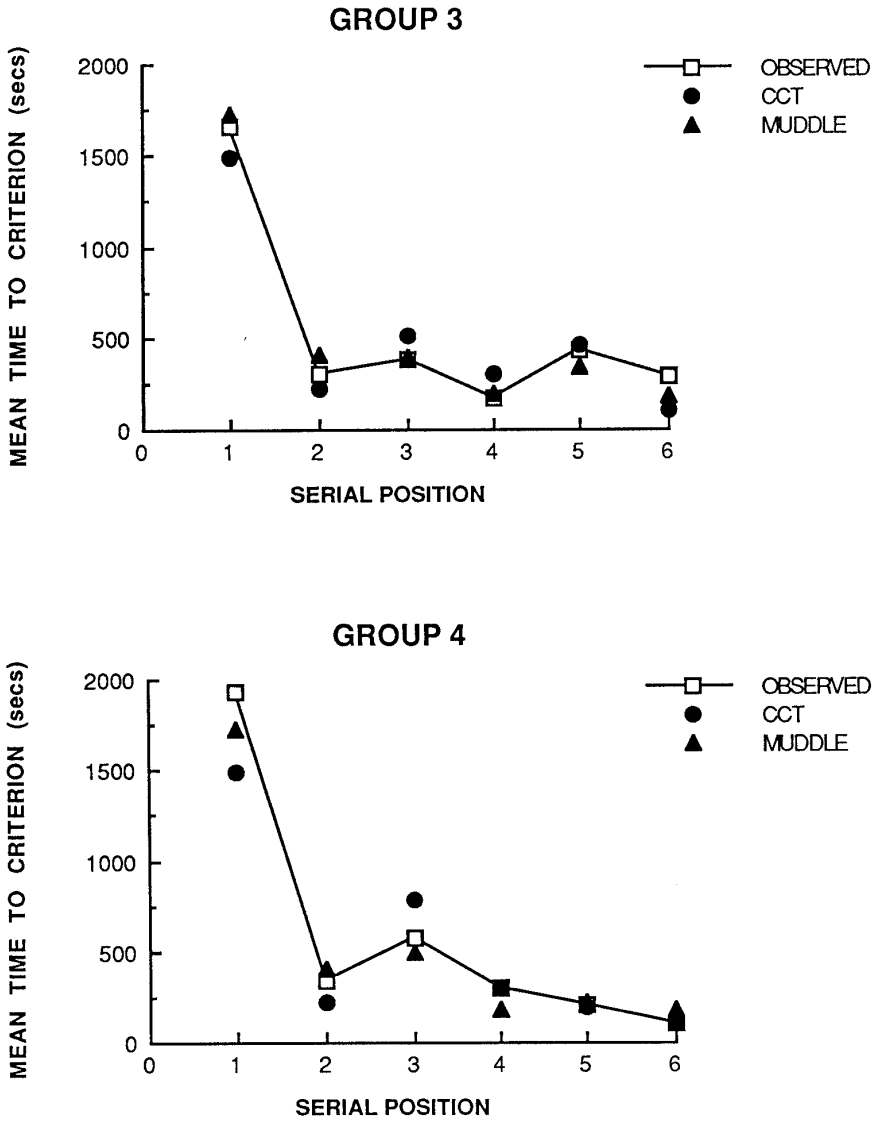

Figure 4-2. (cont.)

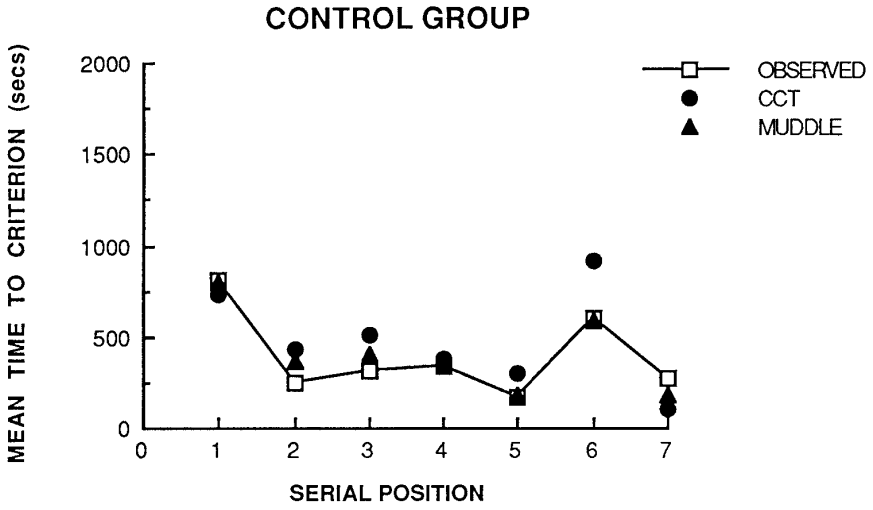

Figure 4-2 (cont.)

Table 4-7.
Regression Analysis on Mean Time to Criterion for Cognitive Complexity Theory
($N = 31$, $R^2 = .876$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	84.13		
New Rules	26.95	.936	204.23

Table 4-8.
Regression Analysis on Mean Time to Criterion for the Muddle Model
($N = 31$, $R^2 = .938$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	177.67		
New Rules	23.14	.968	438.19

CCT's predictor *new rules* and the individual participant means were regressed against individual times to criterion. The fit of these predictors to the individual times to criterion is summarized in Table 4-9. The predictors of the model were able to account for 70 percent of the variance in the individual times to criterion. The observed parameter for number of new rules was 26.77 seconds per rule. This value is similar to the observed value for the mean time to criterion and suggests that the predictor is accounting for the same effects in both analyses. Additionally, it should be noted that the standardized beta weights for each individual's mean time to criterion were less than the standardized beta weights for number of new rules. These weights indicate that new rules was a better predictor of time to criterion than each participant's own mean.

Table 4-10 summarizes the fit of Muddle's predictor *new associations* and individual mean completion times to the dependent variables. These two predictors were able to account for 72 percent of the variance in the individual times to criterion. The observed parameter for new associations was 22.73 seconds per association. Like CCT, the observed parameter was similar to the value observed for the mean time to criterion (23.14 seconds per association). Also, like CCT's fit, the standardized beta weight for new associations indicated that this parameter was a better predictor of each participants performance than their own mean.

Table 4-9.
Regression Analysis on Mean Individual Times to Criterion for CCT
($N = 1078$, $R^2 = .697$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	-404.37		
Ind. Times	.97	.05	322.39
New Rules	26.77	.58	2116.72

Table 4-10.
Regression Analysis on Mean Individual Times to Criterion for the Muddle Model
($N = 1078$, $R^2 = .722$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	-206.57		
Ind. Times	.88	.27	287.52
New Rules	22.73	.79	2415.38

Conclusions

The primary purpose of this chapter was to evaluate the transfer predictions of the Muddle model. This was done by fitting the observed mean time to criterion to the predictors of the model. As Figure 4-2 shows, both models gave excellent fits to the observed transfer results. As a predictor, number of new associations was able to account for 94 percent of variance observed in the mean times to criterion. These fits showed that the predictors of the Muddle models can be used to account for transfer. In addition, the fits of the model provide support for the model's account of transfer.

A second purpose of this chapter was to compare Cognitive Complexity Theory with the Muddle model. Both models gave excellent fits to the observed transfer results should in Figure 4-2. Unfortunately, it is not possible to say that one model gave significantly better account for the data than the other. We can, however, compare the fits of the models. CCT was able to account for 88 percent of the variance in the time to criterion cell means, while Muddle accounted for 94 percent of the variance in the cell means. In addition, the standard errors of the residuals for CCT was 158.76 compared to 112.172 for Muddle. By these measures, Muddle gave the better account for the data, although it cannot be concluded that Muddle gave a significantly better account for the data than CCT.

It is interesting that the predictors of the two models were highly correlated ($r = .92$). One might have expected larger differences in the pattern of predictions of the models. Recall that CCT assumes behavior is controlled by a hierarchical encoding of task goals. On the other hand, Muddle assumes that little encoding of tasks takes place. Instead, the model assumes that the methods needed to perform tasks are selected based on the current system context and the last method solved.

The differences in the models' assumptions about how knowledge is structured do lead to differences in predictions. A comparison of the sum of the predictors for the groups shown in Tables 4-4 and 4-5 reveal differences between the models. Specifically, the models make very different predictions about the amount of knowledge that must be acquired by the control group. The Muddle model predicts that the control group has to acquire 22 to 20 fewer associations than the four experimental groups. In contrast, CCT predicts that the control group has to acquire 8 more new rules than the experimental groups. This difference between the models is a result of there assumptions about how procedures are structured. Muddle's context sensitive selection rules, in combination with its partial matching capability, led it to predict an aggressive transfer of knowledge across the these tasks. CCT's strict decomposition of tasks into goals and subgoals led it to predict that more knowledge was needed to perform the control tasks. CCT made this prediction because the control group was presented with the most varied set of tasks: one prompt answering task, two form-fill tasks, two parameter-entry tasks, and two parameter-entry tasks that required diskette manipulation.

Table 4-11 shows the observed and predicted totals for the mean times to criterion for the 5 groups. Table 4-9 shows that the observed total time for the control group was less than the total time for the experimental groups. A between subjects analysis of variance was performed with groups as the between subjects variable and total time to criterion as the dependent variable. This analysis showed a significant difference between groups $F(4,157) = 2.66$, $p \leq .05$. However, a post hoc comparison between the control groups and the four experimental groups was not significant.

Table 4-11.
Observed and Predicted Mean Total Time to Criterion in Seconds for the Experimental and Control Groups

Group	Observed	CCT Predicted	Muddle Predicted
1	3152	3091	3195
2	3152	3091	3241
3	3234	3091	3241
4	3459	3091	3217
C	2759	3280	2863

Before concluding the present chapter, it should be noted that Muddle's fit to these data does not challenge any of the basic conclusions about transfer drawn from CCT's fit to the data; specifically, Polson's (1988, page 80) conclusion that CCT's fit to the data supported "...the conclusion that transfer was mediated by specific relationships between successive tasks." Muddle's fit to these data also supports this conclusion. Both CCT and Muddle are common element models of transfer and assume that transfer is mediated by shared elements between tasks.

CHAPTER V
EXPERIMENT 2

Purpose

The first goal of this experiment is to look at the effects of inconsistencies on people's ability to learn and retain procedures. It is often claimed that computer applications are difficult to learn and to remember how to use because they contain inconsistencies (Rubenstein and Hersch, 1984; Smith and Mosier, 1986; Polson, 1988). The type of inconsistencies investigated in this experiment are those that result when applications require their users to perform different actions to accomplish similar goals across the application. To successfully accomplish such tasks, users must learn to discriminate when it is appropriate to use one method or another. Polson (1988) has argued that these inconsistencies can: 1) prevent transfer; 2) accelerate forgetting; and 3) interfere with the learning of new methods. This experiment looks at the effects of cueing inconsistencies on people's ability to learn and retain the procedures needed to perform menu tasks.

The second goal of this experiment is to evaluate Muddle's ability to predict the learning and retention of procedures. To prove itself adequate, Muddle must be able to account for both time and error data. It must predict which errors are likely to occur during training and retraining. Specifically, Muddle must be able to predict any transfer or forgetting errors resulting from well-cued and poorly-cued inconsistent methods.

What Are Consistent and Inconsistent Methods?

Figure 5-1 presents a schematic definition for consistent and inconsistent methods. A method is a procedure for accomplishing a goal and is *consistent* if it is the only method associated with a a goal. A method is *inconsistent* if it is one of two or more methods associated with a goal. The representation of inconsistent methods shown in Figure 5-1 makes a distinction between inconsistent methods that will be needed later. Since the existence of an inconsistency implies the existence of more than one method, the various methods need to be differentiated from one another. The inconsistent method appearing with the greatest frequency across tasks will be referred to as the *prototypical method* while the methods appearing less frequently will be referred to as *variant methods*.

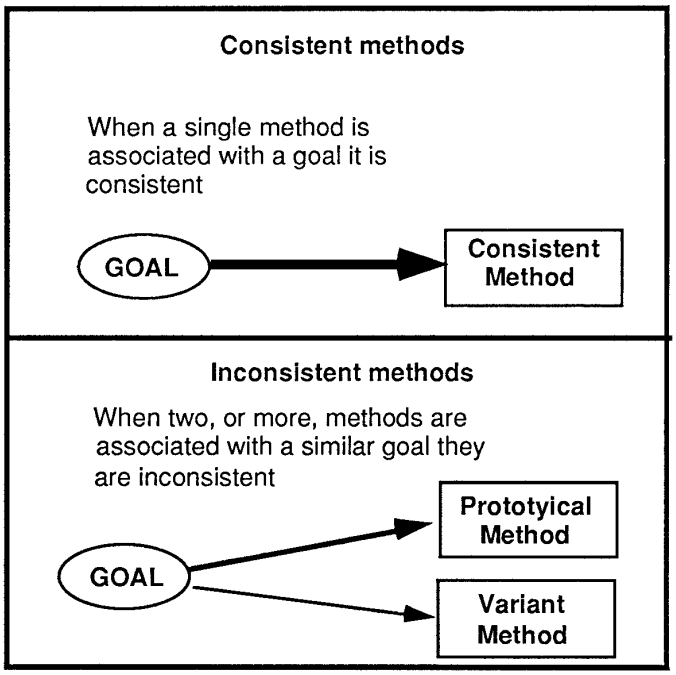

Figure 5-1. Diagram of consistent and inconsistent methods.

Method

Participants

One hundred and four students from the University of Colorado participated in this experiment. Participants served in the experiment in exchange for credit toward fulfillment of a course requirement. Their experience with computers varied from those who had never used a computer before, to those who reporting using up to three text editors or word-processors. None of the participants, however, reported using the word processor simulated in this study.

Hardware and Software

The tasks used in this experiment were performed using a modified version of the simulated menu system described in the previous chapter. The modifications to the simulation involved changing the content of the system's menus and adding several new methods for interacting with menus. These modifications are described below.

Materials

Table 5-1 describes the fourteen different menu tasks created for this experiment. Like the menu tasks presented in the previous chapter, the actions needed to complete these tasks could also be described as a sequence of discrete steps. Again, like the first experiment, these tasks could be classified as *parameter-entry*, *form-fill*, or *prompt-answering* tasks. The *parameter-entry* tasks

required a series of menu traversals until a parameter-entry menu was reached. At this terminal menu participants had to change one of several parameters (e.g., change the line spacing for a document from single to double spaced) and then leave the menu. Completion of the *form-fill* tasks required a series of menu traversals to reach a

Table 5-1.
Task Descriptions for Experiment 2

Task Number	Task Description
Parameter-Entry tasks	
1	Change the default spacing from double to triple.
2	Change the alternate page format so the footer prints on all pages.
3	Change the default page format so the page size in legal size.
4	Paginate a document so its line endings are adjusted.
Form-Fill Tasks	
5	Change a diskette name.
6	Delete a document.
7	Check a document for spelling errors.
8	Print a document.
Prompt-Answering Tasks	
9	Condense a diskette.
10	Load a spelling supplement.
11	Recover a document.
12	Clear a spelling supplement.
13	Condense a document.
14	Print the index for a diskette.

form-fill menu. Upon reaching this menu participants had to fill in the empty slots of the menu's form and then leave the menu. The *prompt-answering* tasks, differed from the *parameter-entry* and *form-fill* tasks. These tasks required a series of menu traversals and until a transition menu was reached. This menu merely prompted users for the information the system needed. After giving the requested information, participants completed the task by leaving the menu. The *prompt-answering* tasks never used or displayed parameter-entry or form-fill menus. These tasks served as filler tasks for a control group.

Two types of methods were created for the *parameter-entry* and the *form-fill* tasks: a *well-cued* method and a *poorly-cued* method. A method was considered to be *well-cued* if the main system prompt reliably indicated it was the appropriate method. In contrast, a method was

considered to be *poorly-cued* if the system prompt did not reliable indicate it was appropriate. For example, a *poorly-cued* method used in this study required participants to press a special-purpose function key (the REQST key) to make a menu transition. The main system prompt ("Type ID letter to choose an ITEM; press ENTER") incorrectly indicated that typing an ID letter was the appropriate method for making a menu transition. Therefore the type-ID prompt was an unreliable indicator of which method was appropriate for making a menu transition.

Design

Participants were assigned to one of eight groups, giving a total of 13 participants in each group. There were two between subjects variables and two within subject variables manipulated in this study. The two between subjects variables were task-type and method. There were four task-type groups: a *consistent* group; a *well-cued inconsistent* group; a *poorly-cued inconsistent* group; and a *control* group. Figure 5-2 presents a schematic description of these four task-groups. The consistent group only learned one method to perform the parameter-entry tasks and one method to perform the form-fill tasks. In contrast, the well-cued and poorly-cued inconsistent groups learned two methods for performing the parameter-entry and form-fill tasks: a prototypical method and a variant method. For the well-cued group the menu system's prompt reliably indicated which method was appropriate, while for the poorly-cued group the system prompt did not reliable indicate the appropriate method. Finally, the control group learned only a single method for performing the parameter-entry and form-fill tasks. This group, however, only performed these methods for one task and served to control for frequency of use of the consistent methods. The second between subjects variable *method* manipulated which method of each method-pair appeared as the prototypical method. This manipulation was included to counter-balance for which method appeared as the prototypical method.

The two within subject variables were session and the serial position of tasks within each session. Participants in this study performed the menu tasks during a training session and during a retraining session.

Procedure

Participants initially learned to perform the eight menu tasks in a training session lasting approximately an hour and a half. Each task was learned using a serial anticipation method. This is the same basic training procedure used in the first experiment. On each task step participants were presented with a display from the simulated menu system. If the participants made a correct response the simulation behaved as the real Displaywriter would. However, if participants made an error, an error message was displayed at the bottom of the menu stating that an error had occurred: this message was designed be be similar in style to the IBM Displaywritermessages. If participants made two consecutive errors they were presented with an error screen which told the participants the current state of the menu system, their immediate goal, and the correct method needed to achieve their goal. The training procedure confined the participants to the correct

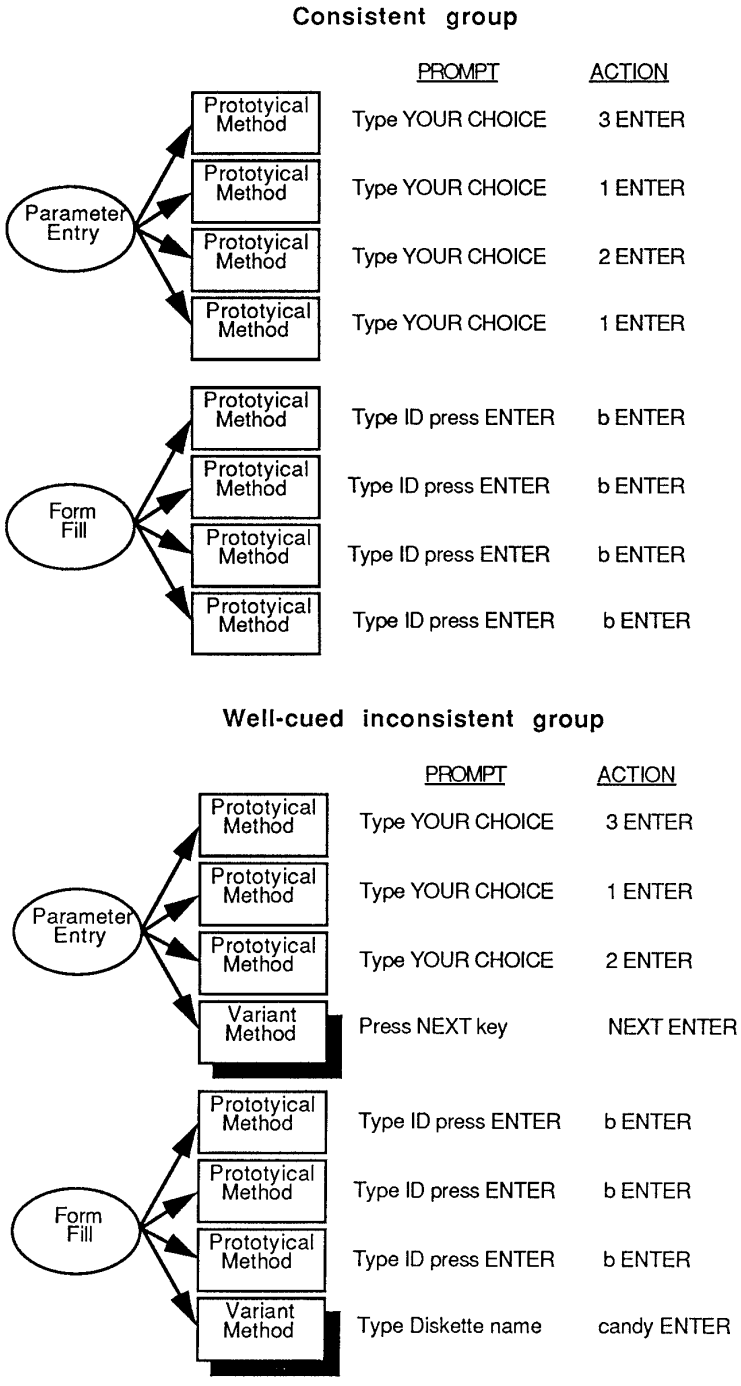

Figure 5-2. Diagram representing the four experimental groups.

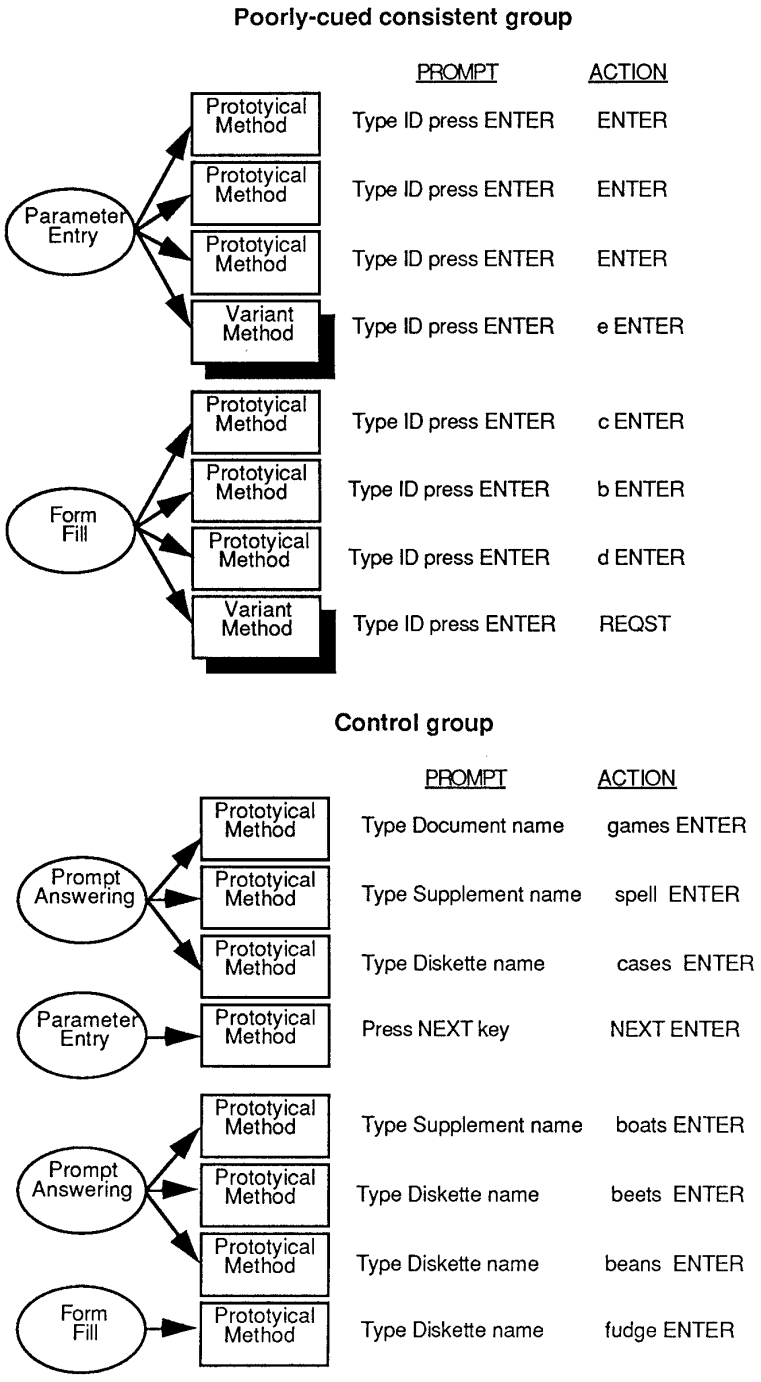

Figure 5-2. (cont.)

solution path for each task. Training for each task continued until participants completed three consecutive correct trials.

During the training session participants in the *consistent* group were trained to perform a block of four parameter-entry tasks followed by a block of four form-fill tasks. These participants learned a single method (i.e., a consistent method) for entering a parameter and a single method for filling in the form-fill menus. Figure 5-2 describes the matching of goals, methods, system prompts, and correct actions of the training tasks. The diagram for the consistent group shows that this group had to perform a single consistent method (typing in a number followed by the "ENTER" key) to accomplish a parameter entry task. In addition, the diagram shows that participants in this group used a single method for specifying form-fill slots (i.e., typing an "ID" letter and pressing ENTER).

Participants in the *well-cued inconsistent* group were also trained to perform a block of four parameter-entry tasks followed by a block of four form-fill tasks. However, these participants had to learn variant methods for performing the last parameter-entry and form-fill tasks (task 4 and task 8). As Figure 5-2 shows, these participants had to learn a variant method for entering a parameter (pressing a "NEXT" key, which cycled through the possible parameter values, and then pressing the "ENTER" key to record their choice). Similarly, to complete the final form-fill task, participants in this group had to learn a variant method for selecting the second slot of the menu (instead of typing an ID-letter to select the second slot, the system skipped this step and prompted for this slot's value after the first slot's value was entered). These methods were well-cued inconsistent methods: well-cued because the system prompt indicated which method was appropriate, inconsistent because more than one method had to learned to accomplish a goal.

Like the consistent group and the well-cued inconsistent groups, participants in the *poorly-cued inconsistent* group were trained to perform a block of parameter-entry tasks followed by a block of form-fill tasks. And, like the well-cued inconsistent group, these participants had to learn a variant method for performing the last parameter-entry task and form-fill task during training. These methods, however, where poorly-cued: They were inconsistent poorly-cued methods because the system's prompt did not reliably indicate which method (prototypical or variant) was appropriate and because participants had learn two methods for accomplishing a goal.

Participants in the *control* group were trained to perform a block of three prompt-answering tasks followed by a single parameter-entry task, these tasks were followed by another block of three prompt-answering tasks and ended with a form-fill task. The purpose of the control group was to control for differences in the amount training between the variant methods and consistent methods. Since the participants in the control group performed only one parameter-entry task and one form-fill task, they learned consistent methods to accomplish these tasks. However, their experience with these methods was comparable to the experiences the inconsistent group participants had with the variant methods.

Participants were brought back approximately twenty-four hours after the training session for a retraining session. During this retraining session participants were asked to perform the same eight tasks they had learned the previous day. The performance criterion for this session, however, was relaxed to one correct trial. In addition, the presentation order of the tasks was changed. The tasks appearing in the fourth and eighth serial position during the training session (the last task of each of the two task blocks) were moved to the first and second serial positions, respectively. These changes were made to minimize the effects of retraining on the performance of these two tasks.

Results and Discussion

The results of the experiment are presented in two sections. The first section presents the effects of experimentally manipulated variables on training and retraining performance. This section presents the results from two analyses of variance and an analysis of the training and retraining errors made by participants. The second section presents the fits to the Muddle model to data from the experiment. This section evaluates Muddle's ability account for the training and retraining results of the experiment.

The Effects of Inconsistencies

Time to criterion. An analysis of variance was performed using time to criterion for each task as the dependent variable. Recall from the first experiment that time to criterion is the amount of time required for a participant to complete a task starting with their initial reading of the instructions to their completion of the last action on their last correct trial. There were two between subject variables (task-type and method) and two within subject variables (session and serial position) in the analyses.

The analysis of variance of times to criterion showed significant main effects for task-type groups ($F(3,96) = 8.21$; $p \leq .01$), session ($F(1,96) = 1271.17$; $p \leq .01$), and serial position ($F(7,96) = 223.21$; $p \leq .01$). In addition, several significant interactions were found. First, the interaction between task-type and serial position was significant, $F(21,672) = 5.98$, $p \leq .01$. This result can be attributed to differences in difficulty between the two sessions and groups. Additionally, the interaction between session and serial position was significant, $F(7,672) = 163.08$, $p \leq .01$. These results can explained by differences in task difficulty and differences in criterion for the two sessions.

The important result was the three-way interaction between task-type, session, and serial position. This interaction was significant, $F(21,672) = 8.27$, $p \leq .01$. Figures 5-3 and 5-4 show the observed mean times to criterion of the four groups for the training and retraining session as a function of serial position, respectively . Figure 5-3 shows the mean task completion times for the training session. The serial positions of interest on this curve are four and eight. These are the serial positions where the inconsistent groups encountered the variant methods. Tukey's *a*

posteriori comparison test was performed on the means in serial position four and showed that the consistent group was significantly faster performing the task than the other three groups ($p \leq .01$). This result shows that consistent methods can reduce task completion times when compared to learning a new variant method or learning to perform the task for the first time. The comparison test also showed that the control group took significantly longer to perform the task than the other groups ($p < .01$). No significant difference between the inconsistent groups was found. A comparison test for serial position eight showed only one significant difference between the groups: The task completion time for the consistent group was significantly faster than the task completion time for the poorly-cued group. The pattern of performance for the inconsistent and control groups suggests that well-cued methods can also increase training time, but not by any more time than would be required to learn to perform the parameter-entry task for the first time.

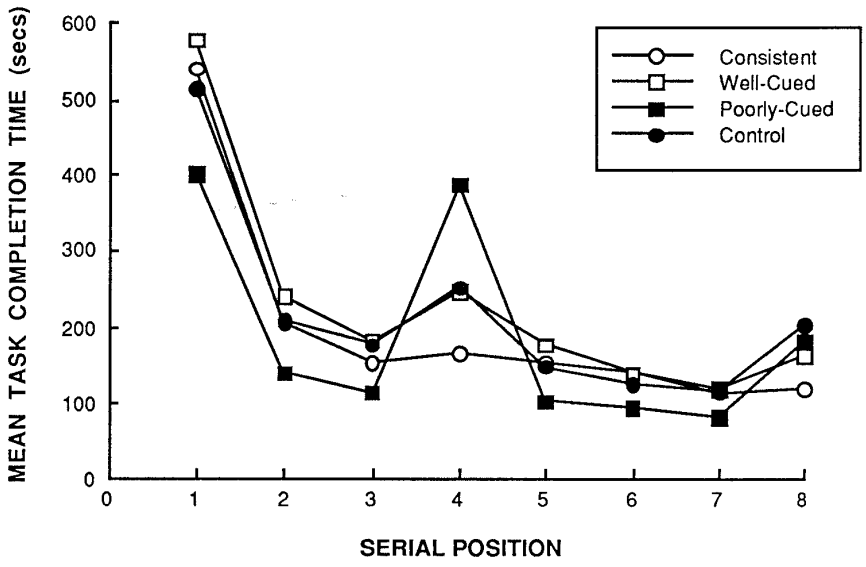

Figure 5-3. Observed mean task completion times for the training Session

Figure 5-4 shows the performance of the four task-type groups for the retraining session plotted as function of serial position. For this session, serial positions one and two are of interest. These were the positions where the inconsistent groups re-encountered the variant inconsistent methods during retraining. Tukey's comparison task, however, showed no significant differences between groups for these two serial positions.

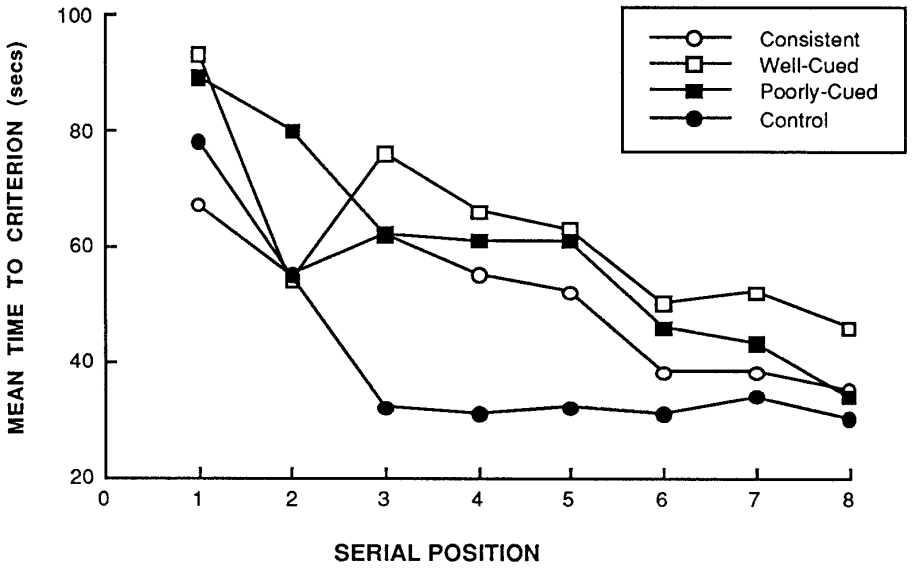

Figure 5-4. Observed mean task completion times for the retraining session

Analysis of errors. An analysis of variance was performed using number of errors made by each participant while performing a task as the dependent variable. Again, the between subjects variables were task-type and method, while the within subject variables were session and serial position. The analysis of errors yielded results comparable to the analysis of criterion times. With the exception of the main effect of task-type, the same main effects and interactions were found. These effects were: session, $F(1,96) = 121.45$, $p \leq .01$; serial position, $F(7,672) = 45.52$, $p \leq .01$; the interaction of task-type with serial position, $F(21,672) = 2.06$, $p \leq .01$; the interaction of session with serial position, $F(7,672) = 36.42$, $p \leq .01$; and the three way interaction between task-type, session, and serial position, $F(21,672) = 2.77$, $p \leq .01$. Again, the interaction of task-type, session, and serial position is of greatest interest. Figures 5-5 and 5-6 shows the mean number of errors made by participants in the task-type groups plotted as a function of serial position. Figure 5-5 shows the pattern of errors for the training session, while Figure 5-6 shows the pattern of errors for the retraining session. An inspection of serial positions four and eight in Figure 5-5 shows that these positions produced the most variability between the task-type groups. Tukey's *a posteriori* comparison test showed that all four task-type groups were significantly different from each other for serial position four ($p \leq .01$). This pattern of errors shows that the control group had the greatest difficulty performing this task. Recall that this is the only parameter-entry task performed by this group. The number of errors made by this group may reflect difficulties participants had performing several steps they had not learned before, including

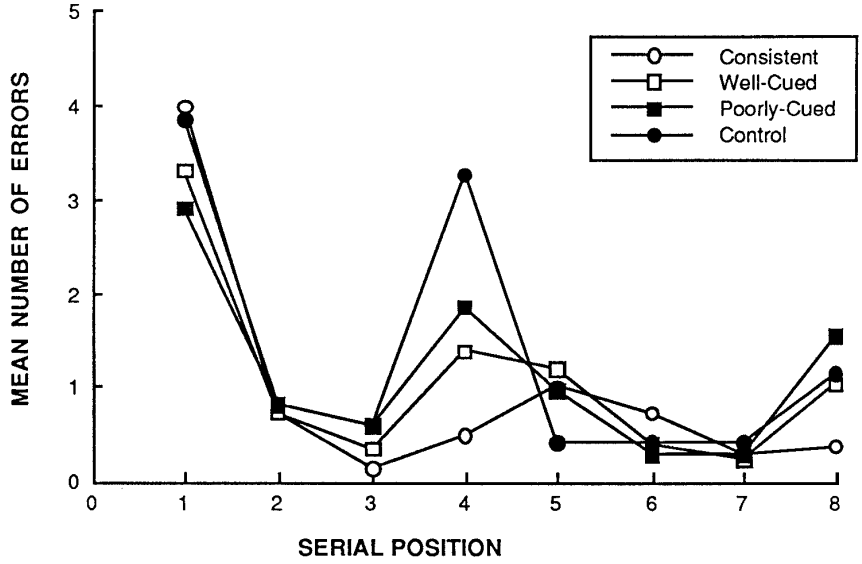

Figure 5-5. Observed mean number of errors for the training session.

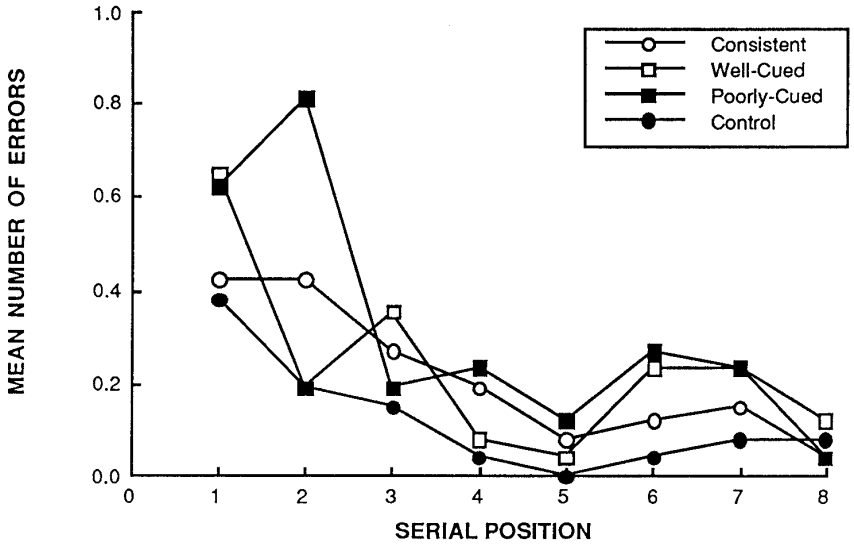

Figure 5-6. Observed mean number of errors for the retraining session.

the step that required using what was the variant method for the well-cued group. The ordering of the two inconsistent groups suggest that the poorly-cued variant methods can lead to more errors than well-cued variant methods.

A comparison test for serial position eight shows that the consistent group made significantly fewer errors than the other three groups ($p \leq .01$), and that the well-cued inconsistent group made significantly fewer errors than the poorly-cued group. Again, this pattern of results suggest that the consistent group had the least amount of difficulty with the tasks and that cueing variant methods can reduce, but not eliminate, the ill-effects of encounters with variant methods.

Figure 5-6 shows the mean number of errors made in each group for the retraining session. The serial positions of interest here are one and two. A Tukey comparison test for serial position one showed no significant difference between the groups, while a comparison test for second serial position showed that the poorly-cued group made significantly more errors than the well-cued and control groups ($p \leq .01$) and the consistent group ($p \leq .05$). These results show that cueing variant methods can reduce retention errors.

The results of the two analyses just reported suggest three conclusions. First, participants in the consistent groups were better off for having learned the parameter-entry and form-fill tasks than were the participants in the inconsistent groups. The consistent group outperformed both inconsistent groups in almost all of the comparisons made. Second, cueing lessens, but does not eliminate, the ill-effects associated with variant methods. In most of the comparisons, the poorly-cued group's performance was worse than the well-cued group's performance. The third conclusion is that the forgetting effects observed cannot not be easily explained by differences in experiences. For the retraining tasks, the control group made the fewest errors for parameter-entry and form-fill tasks. The controls group's experience with these tasks was comparable with the inconsistent group's experiences with the variant methods.

First-attempt errors. Next, analyses of first-attempt errors are presented. A first-attempt error is an error made by a participant on his or her first try of his or her first attempt at a given task step. In the analyses reported below the first attempt errors of participants were analyzed for the steps of the parameter-entry and form-fill tasks where the inconsistent groups encountered the variant methods; in the discussion that follows these steps will be called *critical steps*.

Training errors were defined as first attempt errors made at the critical steps of the prompt-entry or form-fill tasks during the training session. Figure 5-7 shows the number of the participants making a training error for each task-type and method group (a or b) for the parameter-entry and form-fill tasks. On the critical step of the parameter-entry task, a test for significant differences between proportions showed that the consistent task-type groups made significantly fewer errors than the other groups ($p \leq .01$). These results reflect the advantage the consistent groups had over the other groups. Their prior knowledge of the parameter-entry and form-fill tasks could be appropriately applied to this new context. Also, it should be noted that the participants in

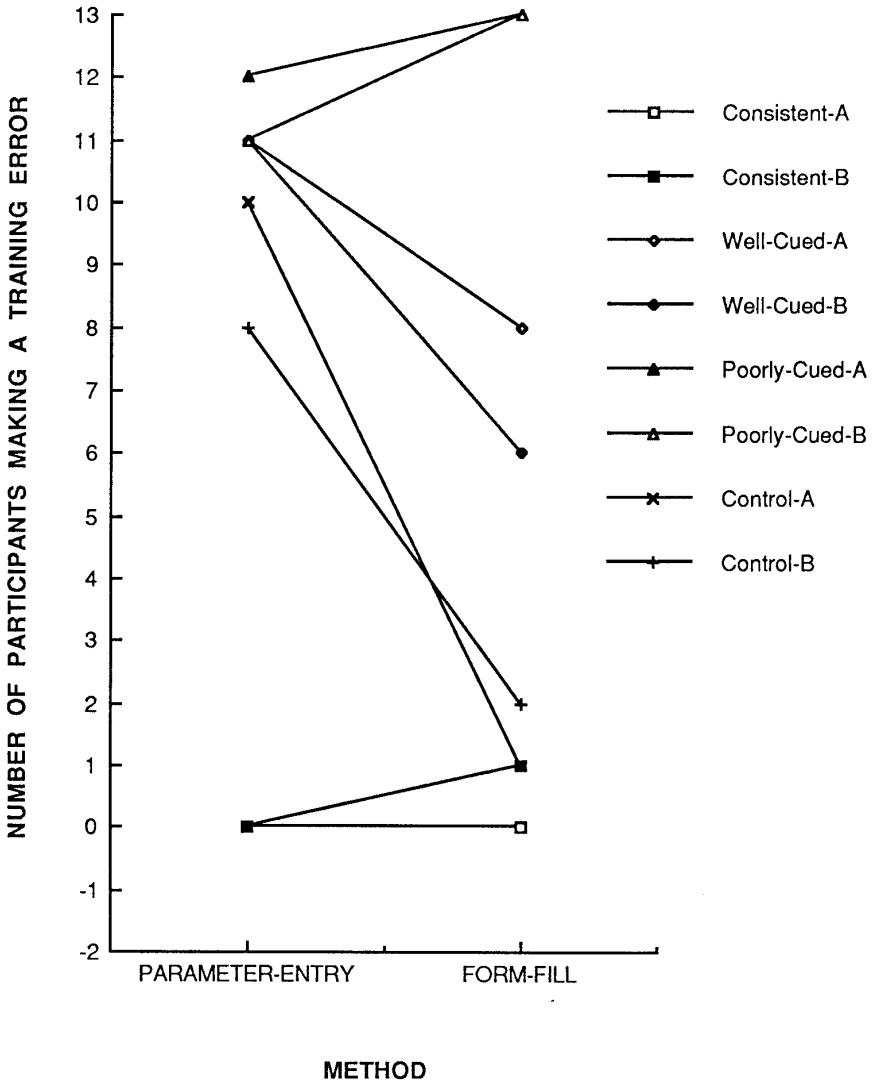

Figure 5-7. Number of participants in each group making a first attempt error at the critical steps of the prompt-entry and form-fill tasks for the training session.

the inconsistent groups were as likely to make training errors when encountering a variant method as the control groups were when encountering a consistent method for the first time.

Tests for significant differences between proportions on the critical step of the form-fill task showed that the consistent groups made significantly fewer errors than the inconsistent groups ($p \leq .05$). The control groups were significantly different from the poorly-cued groups ($p \leq .01$) and the well-cued, method-a group ($p \leq .05$). Finally, the well-cued groups were significantly different than the poorly-cued groups ($p \leq .05$). These results suggest that the consistent groups again where better off for having relevant prior experiences with the form-fill tasks. The comparison of the control groups with the poorly-cued groups suggest that participants in the poorly-cued groups were worse off for having prior experience with the form-fill tasks than was the control group which had no experience with the task. This is also true when the control groups are compared to the well-cued, method-a group. Finally, the training results also show that the negative effects of inconsistencies can be reduced by cueing inconsistencies.Before shifting the discussion to retention errors another analysis of training errors must be reported. An examination of the specific errors made by participants in the training of the inconsistent groups strongly suggest that they were inappropriately transferring their knowledge on the critical steps. Ninety-four percent of the training errors made by the inconsistent groups were made by performing prototypical actions. These results provide strong support for Polson's (1988) claim that inconsistencies lead to inappropriate transfer.

A retention error was defined as a first-attempt error made on a critical step of the parameter-entry or form-fill tasks during the retraining session. Figure 5-8 shows the number of the participants in each task-type and method group making a retention error while performing the critical step of these tasks.

Figure 5-8 shows that inconsistencies can rapidly produce forgetting effects. A test for significant differences between proportions for the parameter-entry tasks showed that the control-method-a group made significantly fewer retention errors than the well-cued-method-a group ($p \leq .01$), the poorly-cued-method-b group ($p \leq .01$), and the poorly-cued-method-a group ($p \leq .05$). These tests also showed that the consistent groups made significantly fewer errors than the well-cued-method-a group ($p \leq .01$) and the poorly-cued-method-b group ($p \leq .05$). Finally, the control-method-b and well-cued-method-b groups were found to be significantly different from the well-cued-method-a group ($p \leq .05$). The most puzzling result found was that the well-cued-method-a group made significantly more errors than the well-cued-method-b group. A trace of the Muddle model suggests that this finding is the result of differences in roles played by system and temporal cues in the selection of each method. The prototypical method for the method-a group required a sequence of three actions be performed in similar contexts. This meant that for prototypical method-a system cues were of little value discriminating between the actions and as a result the temporal cues dominated selection. When this group first encountered the variant method the initial temporal cue was same as the prototypical context. The presence of this cue made an error more likely in for this group than the method-b group where the temporal cues were not as dominant.

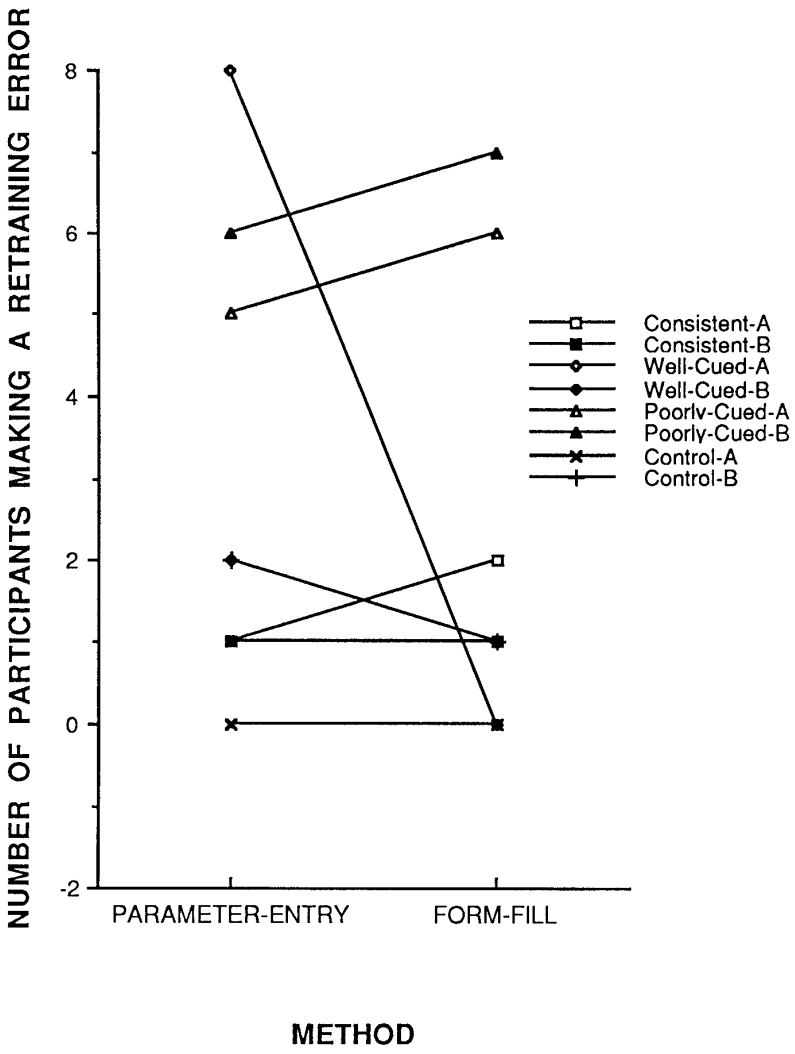

Figure 5-8. Number of participants in each group making a first attempt error at the critical steps of the prompt-entry and form-fill tasks for the retraining session.

Tests for differences between proportions for the form-fill tasks showed that the poorly-cued groups made significantly more retention errors than the other groups ($p \leq .05$). This result shows that the lack of discriminating cues led to a large number of participants making an error in this group. Clearly, these participants had difficulty discriminating when the variant method had to be applied.

Finally, an examination of the specific retention errors made by participants in the inconsistent groups showed that 97 percent of the errors made were made performing the prototypical actions. This result is consistent with an interference theory of forgetting. Specifically, the errors made by the participants are intrusion errors (correct responses made in the wrong context) and suggest that a response competition explanation for forgetting (McGeoch, 1932) can account for these errors.

The results of the analyses of first-attempt errors support the three conclusions from the analyses of times to criterion and number of errors. The conclusion that participants in the consistent group were better off having relevant prior experience with the parameter-entry and form-fill tasks was supported by the observed training and retraining errors. Only two percent of these participants made training errors on the critical steps compared to 82 percent of the participants in the inconsistent groups and 40 percent of the participants in the control groups. Second, cueing variant methods did reduce the ill-effects associated with encounters with variant methods. Sixty-nine percent of the participants in the well-cued groups made training errors compared to 94 percent of the participants in the poorly-cued group. This effect held for retraining errors as well. Ten percent of the well-cued participants made retraining errors on the critical steps compared to 46 percent of the participants in the poorly-cued groups. Finally, the forgetting effects observed were not due to differences in experience with the consistent and variant tasks. This conclusion was supported by a comparison of the performance of the inconsistent groups with the control groups. Participants in control group made fewer retraining errors (six percent of the participants) when compared to the well-cued groups (21 percent) and the poorly-cued groups (46 percent). The experience the control groups had with the parameter-entry and form-fill tasks was comparable to the experience the inconsistent groups had with the variant methods for performing these tasks.

In addition to the conclusions just given, two other conclusions are suggested by the results of the analyses of first attempt errors. First, inconsistencies lead to the inappropriate transfer of knowledge. Ninety-four percent of the training errors made by participants in the inconsistent groups were made by performing prototypical actions. Second, the retraining errors made by participants in the inconsistent groups are consistent with an interference theory of forgetting. Ninety-seven percent of the retraining errors made by these participants were the result of performing prototypical actions. These errors are intrusion errors and suggest that a classical theory of interference can be used to explain these forgetting effects.

Fits to the Model

The acquisition, transfer, and forgetting assumptions of the Muddle model were tested by fitting the observed times to criterion and error data to the predictors of the model. These data should provide a good test of Muddle's learning and retention assumptions. Both the training and retraining data showed large differences in the performance for the task-type groups and for the different serial positions.

Fits to time to criterion. Two fits to times to criterion were made using Muddle's predictors. The first analysis fit the mean time to criterion for each task to Muddle's predictors. The observed and mean predicted mean criterion times for the training session are shown in Figure 5-9 as a function of serial position, while the fits to the retraining session are shown in Figure 5-10. The predicted values were calculated by fitting the Muddle's predictors to the 128 cells means. There were two predictors in the model. The first parameter was a count of the number of new associations added to memory during Muddle's performance of a task. The second, *model errors*, was a count of the number of errors made by Muddle while performing a task. The fit of these predictors to the 128 cell means is summarized in Table 5-2. Muddle's predictors were able to account for 76 percent of the variance in the cell means. Interpreting these parameters in the context of the experiment, the time taken to perform the criterion run and other process not involved in acquisition or retention was 81.69 seconds. The time needed to acquire each new association was 11.36 seconds and the cost in time of an error was 82.29 seconds.

Table 5-2.
Regression Analysis on Mean Time to Criterion for the Muddle Model
($N = 128$, $R^2 = .764$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	81.69		
New Associations	11.36	.698	149.37
Model Errors	82.29	.242	17.90

There were a total of six cases where Muddle's predictions differed significantly from the observed values. The first case occurred for the control method-b group at the fourth serial position in the training session. As Figure 5-9 shows, Muddle underestimated the difficulty the participants in this groups had in acquiring this task. The five remaining significant deviations from observed values occurred at serial position two of the training session. Here, Muddle significantly underestimated the times to criterion for five of the six experimental groups: consistent-method-a; consistent-method-b; well-cued-inconsistent-method-a; well-cued-

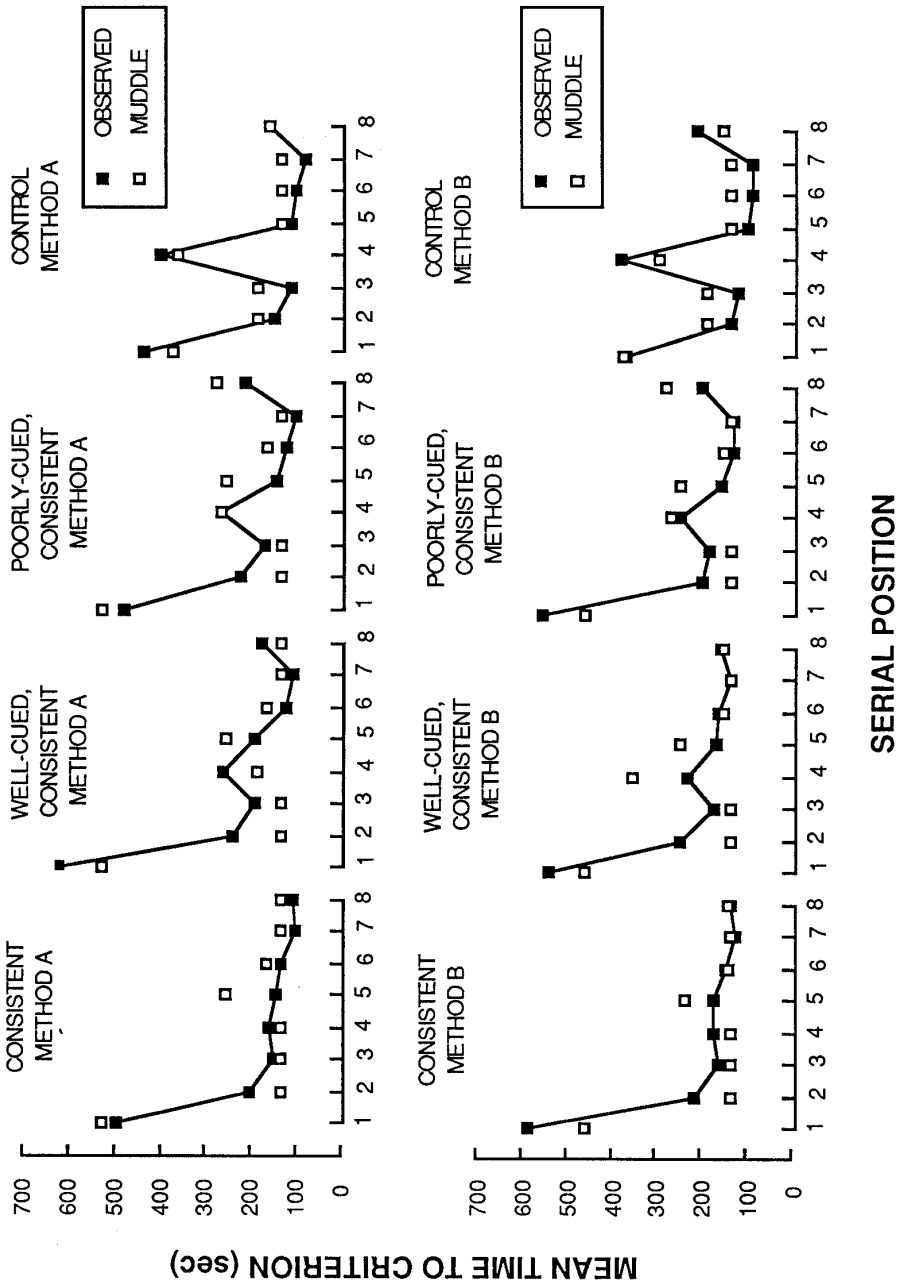

Figure 5-9. Observed and predicted mean task completion times for the training session.

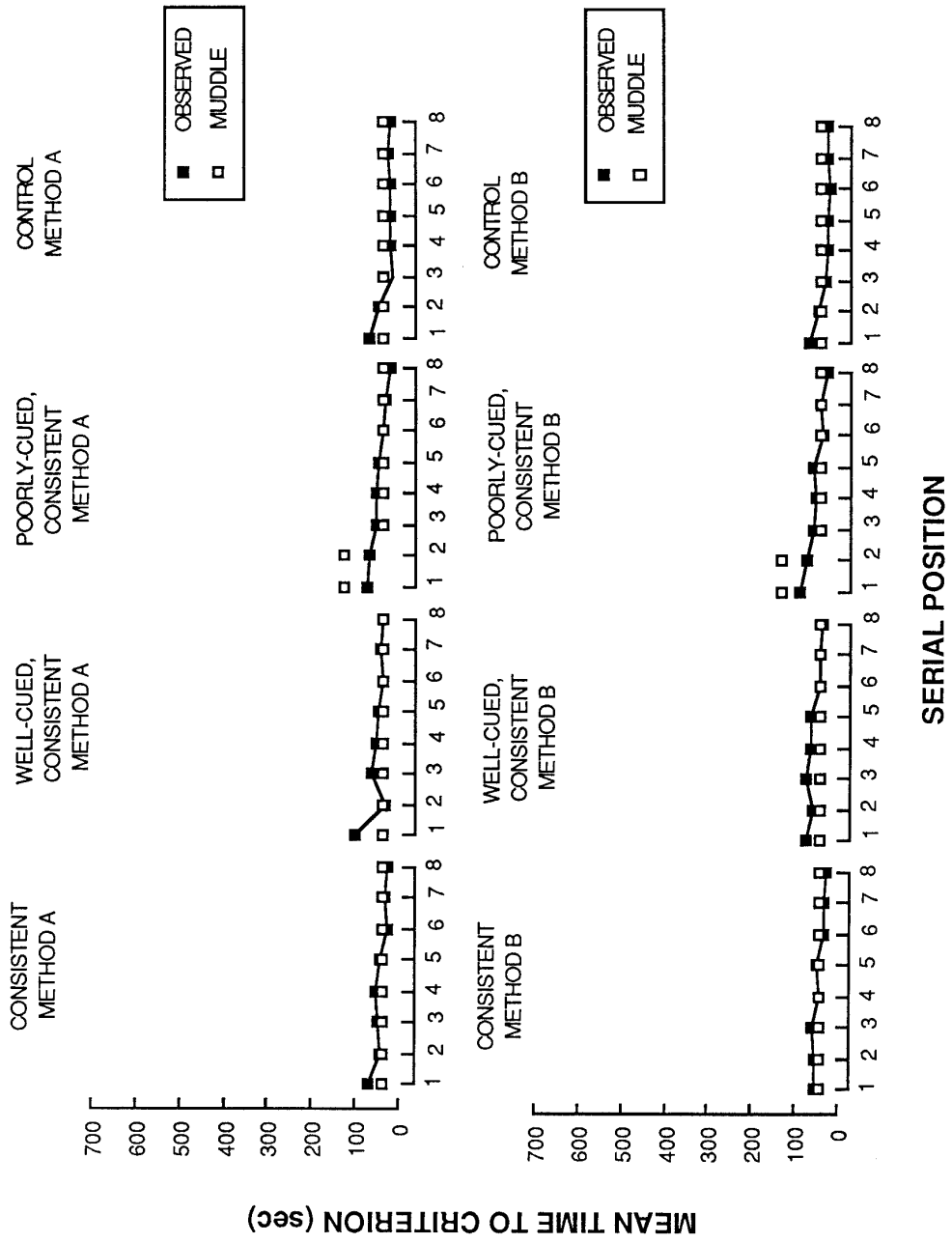

Figure 5-10. Observed and predicted mean task completion times for the retraining session.

inconsistent-method-b; and poorly-cued-inconsistent-method-a. In addition, it should be noted that the predicted value for the remaining experimental group also deviated greatly from the observed value, though this deviation was not significant. These deviations from the observed values are so systematic they merit further exploration.

There are several possible interpretations for Muddle's poor fit to these tasks. One possibility is that the model's generalization mechanism is too aggressive. Recall that Muddle generalizes a problem-solving episode based on a single example. The methods needed to perform the second task are very similar to those used in the first task. Indeed, these tasks are so similar that Muddle does not add any new associations to memory for the second task. If Muddle based its generalizations on more than a single problem-solving episode then, perhaps, it would give a better account for these data. A second possibility is that because Muddle does not create a hierarchical goal structure for tasks, it underestimates the time needed to acquire the second task. It is possible that these deviations could be accounted for by a mechanism which encodes more task-goal information.

A second analysis of times to criterion was performed using the individual times to criterion of participants. Since Muddle does not account for individual differences, each participant's mean time to criterion was included as a parameter to account for individual differences (c.f, Pedhazer, 1982). The results of this analysis are summarized in Table 5-3. Muddle's predictors were able to account for 61 percent of the variance in the individual task completion times. The estimated time for acquiring a new association is 11.36 seconds, while the cost of an error was estimated to be 80.24 seconds. These values are reasonable close to those found for the analysis of the mean task completion times. In addition, the standard values of beta for new associations and model errors were greater than the standard value for each individual's mean task completion time. This result indicates that both errors and new associations were better predictors of performance than each participants mean.

Table 5-3.
Regression Analysis on Individual Times to Criterion for the Muddle Model ($N = 1664$, $R^2 = .610$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	-36.219		
Ind. Mean Time	.904	.183	142.19
New Associations	11.351	.605	128.58
Model Errors	80.238	.205	103.26

Muddle's predictors gave good fits to the observed times to criterion. These fits, however, were not as good as Muddle's fits to times to criterion in the first experiment. In part, this may have due to the differences in criterion between the two sessions in this experiment. Muddle's predictors did not reflect these differences. A clear limitation of Muddle is that it cannot model the performance characteristic that distinguish the two sessions.

The deviations between Muddle's predictions and the observed mean times to criterion suggest that that some of Muddle's assumptions about encoding are not correct. The observed deviations were systematic and suggest that either Muddle's generalization mechanisms are too aggressive or that Muddle's assumption that little task-goal structure is encoded is wrong.

Fits of errors. Next, the fits of the Muddle model to the observed error data are presented. Three fits of the model to data are presented. The first fits Muddle's predictors to the observed number of participants making first-attempt errors during the training and retraining session. The second analysis fits Muddle's predictors to the number of errors made by participants for each task. Finally, the third analysis fits the model's predictor to the number of errors made by individual participants for each task.

The total number of errors made by participants on their first try of their first attempt were analyzed using new associations and model errors as predictors. The observed and predicted values from this analysis are shown in Figures 5-11 and 5-12 as a function of serial position. The model's predictors were able to account for 80 percent of the variance in the 128 totals. The fit of Muddle's predictors to first-attempt errors is summarized in Table 5-4. Table 5-4 shows the number of new associations was the best predictor of errors. The observed number of errors made for each new association was .56 errors. The observed number of errors made for each error made by the model was 7.78 errors. This parameter suggest that the errors made by the model was a good predictor of tasks where participants made a large number of errors.

There were seven significant differences between the predicted and the observed number of first-attempt errors. All seven differences occurred for tasks in the training session. Six of the seven observed differences occurred for serial positions one, four, and eight for the well-cued-method-a group, and for the poorly-cued-method-b group. The other difference occurred in serial position two for the control-method-a group. Five of the six differences were underpredictions of the errors made by participants. Two of the five underpredictions are the result of Muddle's not making errors when it encountered a well-cued inconsistent method. The failure of the model to predict these errors suggest that participants are not as sensitive to the cues presented by the system as Muddle. This may be because participants have encoded task-goal information that lessens their sensitivity to cues provided by system states.

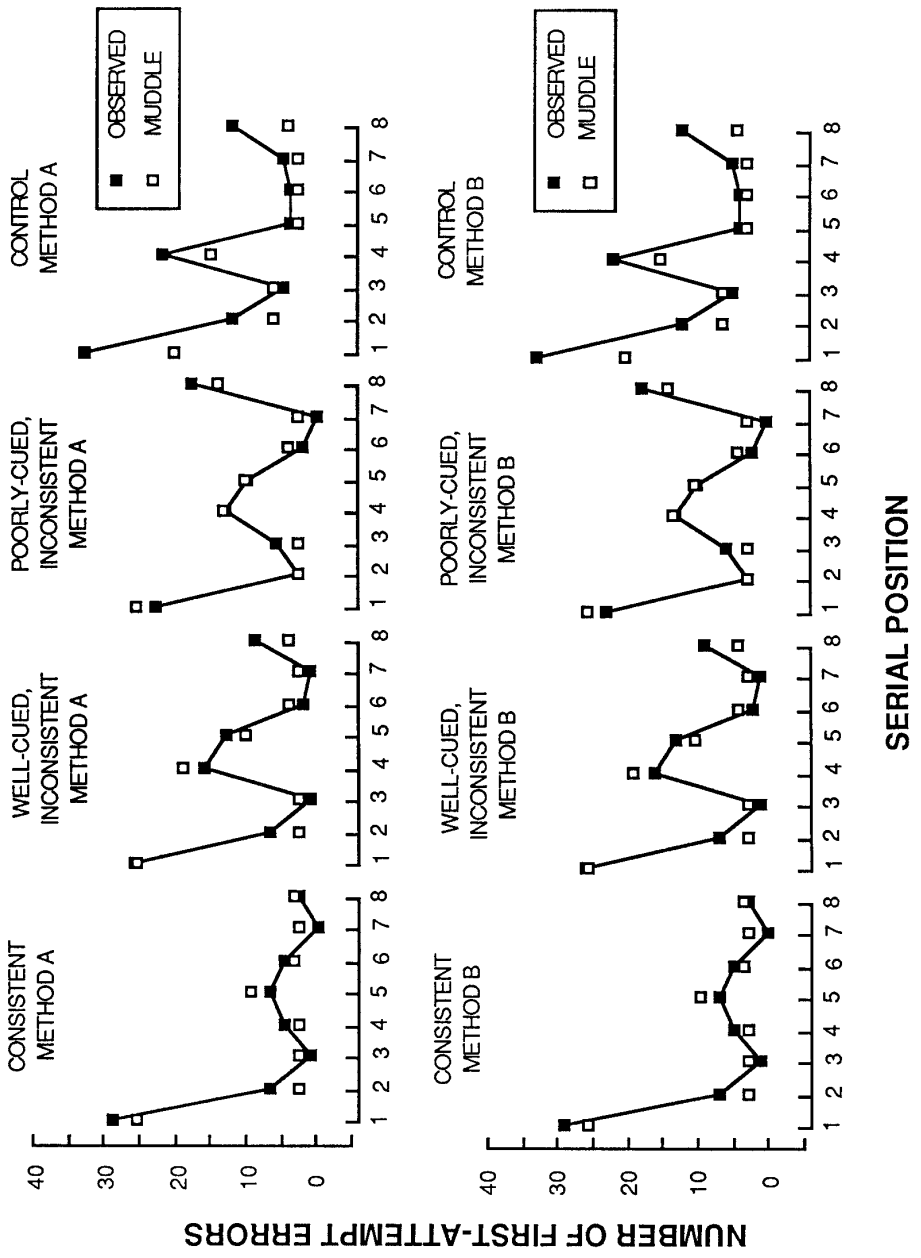

Figure 5-11. Observed and predicted first-attempt errors for the training session.

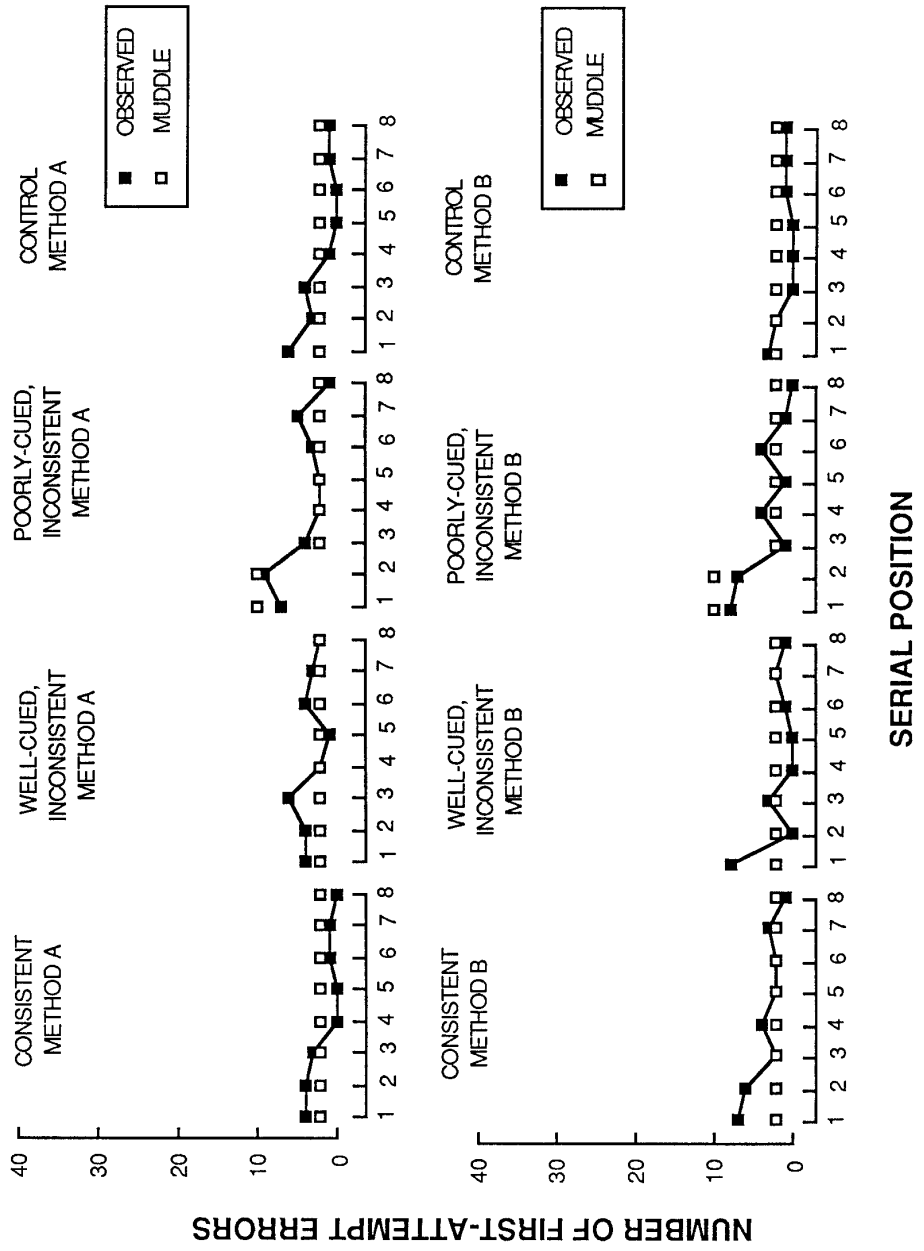

Figure 5-12. Observed and predicted first-attempt errors for the retraining session.

Table 5-4.
Regression Analysis on Total Number of First-Attempt Errors for the Muddle Model (N = 128, R^2 = .804)

Variable	Coef.	Std. Coeff.	Partial F
Constant	2.980		
New Associations	.557	.590	128.58
Model Errors	7.779	.394	57.27

Next, Muddle's fit to the total number of errors made by participants is presented. The total number of errors are highly correlated with the number of participants making first-attempt errors (r = .871). The fit of Muddle's predictors to these data is given in Table 5-5. The predictors of the model were able to account for 77 percent of variance in the 128 mean number of errors. Not surprisingly, the observed parameters are higher than those found for first-attempt errors because total number of errors includes errors made on all attempts and tries of the tasks. However, the standard beta weights for these parameters are similar in the two analyses.

Table 5-5.
Regression Analysis on Error Totals for the Muddle Model (N = 128, R^2 = .77)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	3.79		
New Associations	.934	.568	103.24
Model Errors	13.66	.397	50.48

The final analyses presented are fits to the number of errors made by individual participants for each task. As was done with the fits to the individual times to criterion, the average number of errors made by each participant was included to remove variance due to individual differences. The results of this analysis are summarized in Table 5-6. Muddle's predictors accounted for only 38 percent of the variance in the individual error totals. In addition, the standardized beta weights for the predictors in the model are not as good predictors of errors as each individuals mean number of errors.

Muddle gave good fits to the data. There is, however, room for improvement. First, Muddle cannot account for all the errors observed in this study. In particular, Muddle cannot predict menus errors made by users incorrectly choosing items they believe are alternative

descriptions for the tasks they are performing. The second limitation is the result of using model errors as a predictor. Recall that model errors record the where Muddle made errors performing the tasks. Model errors inadequately reflects Muddle's ability to predict interference effects. It should be possible from tracing Muddle's retrieval mechanism to produce a predictor that reflects the relative strengths of competing methods. These strengths may be better predictors of errors than *model errors*.

Table 5-6.
Regression Analysis on Individual Error Totals for the Muddle Model
($N = 1664$, $R^2 = .38$)

Variable	Coeff.	Std. Coeff.	Partial F
Constant	-.38		
Ind. Mean Errors	.07	.346	185.78
New Associations	1.006	.227	80.06
Model Errors	.988	.321	274.95

Conclusions

The present experiment had two goals. The first goal was to investigate the effects of well-cued and poorly-cued inconsistencies on transfer and retention. The second goal was to test the transfer and forgetting assumptions of the Muddle model.

The Effects of Well-Cued and Poorly-Cued Inconsistencies

There were four important empirical results from this experiment. First, inconsistencies can lead to the inappropriate transfer of knowledge as Polson (1988) has claimed. Most of the participants serving in the inconsistent groups made training errors when they encountered variant-inconsistent methods for the first time. Ninety-four percent of these training errors were the result of performing prototypical methods. The second important empirical result was that cueing does reduce the ill-effects associated with variant methods. In both the training and the retraining sessions the poorly-cued groups made more first-attempt errors than the well-cued groups. This result suggests that if the addition of variant methods to a system is unavoidable, designers should make sure these methods are well-cued by system states. The third important empirical result is that inconsistencies can lead to rapid, systematic forgetting. Thirty-four percent of the participants made a retraining error when performing variant methods they had performed a day earlier. This contrasts with thirteen percent of the participants in the consistent and control groups making a retraining error on the comparable step. The fourth, and perhaps most important, is the result that 97 percent of the retraining errors made by the inconsistent groups were made by inappropriately

performing prototypical methods. These errors can be classified as intrusion errors and their presence is consistent with a response competition model of forgetting proposed by McGeoch (1932). This is a form of an interference theory of forgetting and strongly suggest that classical interference theory can be applied to skill retention.

Muddle's Fits to the Data

Muddle's predictors gave respectable fits to the time and error data from the present experiment. These fits, however, were not as good as had been hoped. Muddle's fit to times to criterion for this experiment was less spectacular than its fit to the training times from the first experiment. There are several reason why Muddle may not have performed as well in the present experiment. The times to criterion used in this experiment were taken from two sessions. These sessions set different performance criteria for the completion of tasks. Muddle's predictors did not take these differences into account and this may have weaken Muddle's ability to account for these data. Second, the deviations of Muddle's predictors from the observed mean times to criterion suggest that Muddle's encoding assumptions may not be correct.

Muddle's fit to the error data was respectable but not as good was hoped. There are several reasons why this was true. First, Muddle is not able to account for errors that result when people do not understand system or task terms, when they incorrectly recognize a menu term as synonym for their goal, and when they cannot recognize general descriptions of tasks that encompass the task they wish to perform. These errors occurred during the experiment and added variance to the data that Muddle could not account for. Second, there is evidence from the analysis of differences between the observed and predicted first-attempt errors that Muddle is more sensitive to system context than were the experiment participants. This suggest that Muddle's performance might be improved by encoding more knowledge of task goals. Finally, the Muddle's predictor of errors, *model errors*, was too coarse. This predictor was a dichotomous predictor of errors. It should be possible to construct a more sensitive predictor of errors by examining Muddle's memory retrieval mechanism.

In closing, while Muddle's fits to the data were not as good as hoped for they were respectable and there is every reason to believe that Muddle's account for these data could be improved. The major limitations of Muddle can be remedied by changing its encoding assumptions so that more task knowledge is encoded and by using a trace of retrieval to produce a better predictor for errors. It is important to recognize that these limitations are not the result of the fundamental architectural assumptions underlying Muddle, but are mainly the result of its extreme system-centered encoding assumptions and its gross predictor of errors.

CHAPTER VI
CONCLUSIONS

The conclusions of the dissertation can be divided into empirical and theoretical conclusions. In the discussion that follows the empirical conclusions will be presented first along with their implications for human-computer interaction and cognitive psychologists. After the empirical results are presented, the theoretical results of this dissertation will be presented. This discussion will center on the Muddle model's performance, specifically its goals, successes, failures, and uses.

Empirical Conclusions

There are a number of important empirical conclusions that can be drawn from this dissertation; first, procedural knowledge can be rapidly and systematically forgotten; second, an interference explanation for procedural forgetting is consistent with forgetting effects observed in this study; third, inconsistent methods *can* lead to the inappropriate transfer of knowledge; and fourth, cueing inconsistent methods can lessen the ill-effects associated with variant inconsistent methods.

Procedures Can be Rapidly and Systematically Forgotten

The results of the retraining session of experiment two replicate Engelbeck's (1986) findings that procedural knowledge can be rapidly and systematically forgotten. The participants in the inconsistent groups made more errors performing the steps of tasks requiring variant methods than the consistent and control groups. On average 34 percent of the participants in the inconsistent groups of experiment two made an error when they had to employ a variant method, compared to four percent for the control groups and 10 percent for the consistent groups. This result was obtained only one day after these participants were trained to perform the same tasks to a criterion of three consecutive correct recitations.

This demonstration of rapid, systematic forgetting effects for skills is important for models of skill. These models, as Anderson (1987) argued, are not constrained enough by current data. These models have not been applied to forgetting data and would profit from the constraints provided by systematic effects present in these data.

Interference Theory Can be Used to Explain Procedural Forgetting

The second empirical result from this dissertation is that to account for the forgetting of skills cognitive models of skill, one would have to be able to predict interference effects. The procedural forgetting results of experiment two strongly suggest that an interference model of forgetting can be applied to the forgetting of skills. In experiment two, 94 percent of the retraining errors made by the inconsistent groups on the variant steps were the result of performing the

prototypical actions. This result is just what one would expect from an interference theory of forgetting.

Inconsistencies Lead to the Inappropriate Transfer of Knowledge

Inconsistency is one of the watchwords of human-computer interaction. Numerous design guides and texts stress that designers should avoid inconsistencies. Specifically, Polson (1988) has claimed that inconsistent methods can lead to the inappropriate transfer of knowledge. The transfer errors observed in experiment two support Polson's claim. When the participants in the inconsistent groups encountered steps where a variant method had to be employed, 82 percent made errors. More importantly, 94 percent of the errors made by these participants were made by performing prototypical actions. The implications of these results should come as no surprise: As the guidelines and texts suggests, inconsistent methods should be avoided when possible.

Cueing Variant-Inconsistencies Reduces Errors

One way designers have tried to minimize the effects of inconsistencies has been to alert users to their presence. These attempts may be last ditch efforts on the part of designer to help users, or they may be performed with the belief that they will prevent users from making errors. In either case, the results of experiment two show that cueing variant methods does not prevent users from making errors but does lessen their associated ill-effects. In both the training and the retraining sessions of experiment two the poorly-cued groups made more first-attempt errors than did the well-cued groups. This result suggests that if designers cannot avoid adding variant methods to a system they should assure that system states cue these methods.

Theoretical Conclusions

The theoretical conclusions from this dissertation center around the performance of the Muddle model. Experiment one tested the transfer assumptions of the Muddle model. Here, Muddle gave a good account for the observed transfer results. Muddle accounted for 94 percent of the variance in the observed task completion times. Muddle's account for the performance on these task was as good as those made by Cognitive Complexity Theory.

Muddle's account for the training and retraining data from experiment two were respectable, but not as impressive as the fits obtained in experiment one. The fits of Muddle's predictors to these data reveled a number of weaknesses in Muddle. First, Muddle had no way to account for the differences in performance criterion between the training and retraining sessions. Currently contrived performance measures could be derived from the model. The long term solution to this problem would be to extend Muddle so it performs each task until it reaches the experimenter set performance criterion. Currently, Muddle only performs a task once attempting to apply its existing knowledge to task and counting the number of associations that must be added to perform the task.

Second, Muddles predictor of errors was inadequately coarse. Muddle's predictor *model errors* only models the majority of users. This is inadequate. Even perfect predictions of where a

majority of participants would make an error would be inadequate. Such a model would only predict eight of the sixteen groups are likely to make training errors in experiment two, and would also predict that only two of the sixteen groups would be likely to make retraining errors. As a theoretical model and as a design tool, Muddle needs to be able predict when a significantly small number of users are likely to make errors.

Third, there was evidence that Muddle's assumptions about how users encode system and task state information was too extreme. The significant deviations of Muddle's predictions of number of errors and observed number of errors for experiment two suggest that Muddle is more sensitive to cues provided by the system than participants are. These deviations from observed values suggest that task state information should be added to Muddle. This conclusion is also supported by the errors Muddle made while learning the diskette-manipulation tasks in experiment one. Recall, participants in these tasks had to load and unload diskettes in the right slot of a simulated diskette unit, and for the work diskette duplication tasks participants had to unload the diskette from the left slot of the unit and then load another diskette in its place. The two diskette duplication steps appeared in different task contexts and participants had few problems discriminating these contexts once learned. Muddle, however, did not encode this task-context knowledge and as a result made errors because it could not discriminate between these two diskette manipulation tasks.

Despite Muddle's limitations it should be considered as a success. It provided a respectable account for both learning and retention data. Its general architecture was also successful. Muddle was able to store, retrieve, and execute procedures stored in an associative memory. None of the limitations of Muddle were a result of its architecture. Muddle's limitations were mainly the result of its extreme system-centered view of encoding and its course prediction of errors.

What is the Muddle model good for? Norman (1988) makes a distinction between knowledge in the world and knowledge in the head. Muddle is a model that relies on the knowledge available in the world to guide it behavior. The results of the two experiments have suggested that Muddle is more sensitive to system states than people are. This sensitivity of Muddle to system states suggest that it would provide a good engineering model for designing machines and applications that people can use without reading manuals or attending training courses. This is a vast and growing proportion of the machines and automated services used by people: including automated tellers, kitchen appliances, computerized card catalogs, and online phone services. Muddle would encourage designers to state what problem solving knowledge users need to operate a system and would show them where system cues mislead users. In addition, Muddle would be able to predict where users are likely inappropriately to transfer their knowledge or to forget which method needs to be applied to accomplish a task.

REFERENCES

Anderson, J.R. (1982). Acquisition of cognitive skill. *Psychological Review, 89*, 369-406.

Anderson, J.R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Anderson, J.R. (1986). Knowledge compilation: The general learning mechanism. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Volume II, Los Altos, CA: Morgan Kaufmann.

Anderson, J.R. (1987a). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review. 94(2)*, 192-210.

Anderson, J.R. (1987b). Methodologies for studying human knowledge. *Behavioral and Brain Sciences, 10*, 467-505.

Anderson, J.R. (1987c). Causal analysis and inductive learning. In P. Langley (Ed.), *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 288-299), Los Altos, CA: Morgan Kaufmann.

Anderson, J.R., & Bower, G.H. (1973). *Human associative memory.* Washington DC: Winston.

Anderson, J.R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8*, 87-129.

Anderson, J.R., & Thompson, R. (1986). Use of analogy in a production system architecture. Paper presented at the Illinois Workshop on Similarity and Analogy, Champain-Urbana, IL.

Anderson, R.C., & Pitchert, J.W. (1978). Recall of previously unrecallable information following a shift in perspective, *Journal of Verbal Learning and Verbal Behavior, 17*, 1-12.

Bovair, S., Kieras, D.E., & Polson, P.G. (1988). *The acquisition and performance of text editing skill: A production system analysis.* Technical Report No. 28: University of Michigan.

Bower, G.H., & Hilgard, E.R. (1981). *Theories of learning.* Englewood Cliffs, NJ: Prentice-Hall.

Card, S. K., Moran, T.K., & Newell, A. (1980). Computer text-editing: An information processing analysis of a routine cognitive skill. *Cognitive Psychology, 12*, 32-74.

Card, S. K., Moran, T.K., & Newell, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, New Jersey: Erlbaum.

Carroll, J.M., & Rosson, M.B. (1987). The paradox of the active user. In J.M. Carroll (Ed.), *Interfacing thought: Cognitive aspects of human-computer interaction.* Cambridge MA: Bradford Books/MIT Press.

Chase, W.G., & Simon, H.A. (1973). The mind's eye in chess. In W.G. Chase (Ed.), *Visual Information Processing.* New York: Academic Press.

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.

Crossman, E.R.F.W. (1959). A theory of the acquisition of speed-skill. *Ergonomics, 2*, 153-166.

Davis, R. (1980). Meta-rules: Reasoning about control. *Artificial Intelligence, 15*, 179-222.

Engelbeck, G. (1986). *Exceptions to Generalizations: Implications for Formal Models of Human-Computer Interaction.* Masters Thesis: University of Colorado.

Engelbeck, G., & Polson, P.G. (1987). An interference theory explanation of retention errors. *Poster presented at CHI+GI'87.*

Ericsson, K.A., & Crutcher, R.J. (1988). *Long-tern retention of sequentially organized information, knowledge, and skills: An empirical review* (Institute of Cognitive Science Tech. Report No. 88-14). Boulder, CO: University of Colorado.

Farr, M.J. (1987). *The Long-Term Retention of Knowledge and Skills: A Cognitive and Instructional Perspective.* New York: Springer-Verlag.

Field, A.E, & Harris, C.L. (1986). The information business. *Business Week, August 25*, 56-90.

Fietelson, J., & Stefik, M. (1977). *A case study of the reasoning in a genetics experiment* (Heuristic Programming Project, Working Paper 77-18). Stanford, CA: Stanford University.

Fitts, P.M., & Posner, M.I. (1967). *Human Performance.* Belmont CA: Brooks Cole.

Foltz, P.W., Davies, S.E., Polson, P.G., & Kieras, D.E. (1988). Transfer between menu systems. In E. Soloway, D. Frye, & S.B. Sheppard (Eds.), *Proceedings CHI'88 Human Factors in Computing* (pp. 107-112). New York: Association for Computing Machinery.

Furnas, G.W., Landauer, T.K., Gomez, L.M., & Dumais, S.T. (1984). Statistical semantics: Analysis of the potential performance of keyword information systems. In J.C. Thomas, & M.L. Schneider (Eds.), *Human Factors in Computer Systems.* Norwood, NJ: Ablex.

Genesereth, M. (1983). An overview of meta-level architecture. *Proceedings of the Third Annual National Conference on Artificial Intelligence* (pp. 119-123), Los Altos, CA: Morgan Kaufmann.

Hanau, P.R., & Lenorovitz, D.R. (1980). Prototyping and simulation tools for user/computer dialog design. In J.J. Thomas (Ed.),*SIGGRAPH'80 Conference Proceedings* (pp. 271-278). New York: Association for Computing Machinery.

Hurlock, R.E. & Montague, W.E. (1982). *Skill retention and its implications for navy tasks: An analytical review* (NPRDC Special Report No. 82-21). San Diego, CA: Navy Personnel Research and Development Center.

Jacob, R.J.K. (1983). Using formal specifications in the design of the human-computer interface. *Communications of the ACM, 26(4)*, 259-264.

Jeffries, R., Turner, A.A., Polson, P.G., & Atwood, M.E. (1981). The processes involved in designing software. In J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

Kieras, D.E., & Bovair, S. (1986). The acquisition of procedures from text: A production system analysis of transfer of training. *Journal of Memory and Language. 25*, 507-524.

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

Kintsch, W. (1972). *Learning, memory, and conceptual processes.* New York: Wiley.

Kintsch, W. (1982). Memory for text. In A. Flammer & W. Kintsch (Eds.), *Discourse Processing.* Amsterdam: North-Holland.

Kintsch, W., & Keenan, J.M. (1973). Reading rate and retention as a function of the number of propositions in base structure of sentences. *Cognitive Psychology, 5*, 257-274.

Kintsch W. & Mannes, S.M. (1987). *Generating scripts from memory* (Institute of Cognitive Science Tech. Report No. 87-3). Boulder, CO: University of Colorado.

Kintsch, W., & vanDijk, T.A. (1978). Toward a model of text comprehension and reproduction. *Psychological Review, 85*, 363-394.

Kline, P. (1981). The superiority of relative criteria in partial matching and generalization. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp 296-303), Los Altos, CA: Morgan Kaufmann.

Kolers, P.A. (1979). Reading a year later. *Journal of Experimental Psychology: Human Learning and Memory, 2*, 554-565.

Laird, J.E., Newell, A., and Rosenbloom, P.S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33*, 1-64.

Laird, J.E., Rosenbloon, P.S., & Newell, A. (1986). *Universal subgoaling and chunking: The automatic generation and learning of goal hierarchies.* Boston, MA: Kluwer Academic Publishers.

Larkin, J.H. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. In J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

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

Lesgold, A.M., Feltovich, P.J, Glaser, R., & Wang, Y. (1981). *The acquisition of perceptual diagnostic skill in radiology* (Learning Research and Development Center Technical Report No. PDS-1). Pittsburg, PA: University of Pittsburg.

Lewis, C.H. (1978). *Production System Models of Practice Effects*, Ph.D. dissertation: University of Michigan.

Lewis, C.H. (1982). *Composition of Productions* (Technical Report No. 42959) Yorktown Heights, NY: IBM Watson Research Center.

Lewis, C.H. (1986). A model of mental model construction. In M. Mantei & P. Orbeton (Eds.), *Proceedings CHI'86 Human Factors in Computing* (pp 306-313). New York: Association for Computing Machinery.

Lewis, C.H. (1988). Why and how to learn why: Analysis-based generalization of procedures. *Cognitive Science*, 12, 211-256.

Lewis, C.H., & Mack, R.L. (1982). Learning to use a word processing system: Evidence from thinking aloud protocols. *Proceedings of the Conference on Human Factors in Computing Systems* (pp 387-392), New York: Association for Computing Machinery.

Lewis, M.W., & Anderson, J.R. (1985). Discrimination of operator schemata in problem solving: Learning from examples. *Cognitive Psychology, 17*, 26-65.

Luce, R.D. (1959). *Individual choice behavior.* New York: Wiley.

Mack, R.L., Lewis, C., & Carroll, J.M. (1983). Learning to use word processors: Problems and prospects. *ACM Transactions on Office Information Systems, 1*, 254-271.

McGeoch, J.A. (1932). Forgetting and the law of disuse. *Psychological Review, 39*, 352-370.

McGeoch, J.A. (1942). *The Psychology of Human Learning: An Introduction.* New York: Longmans, Green and Co.

Miller, J.R., & Kintsch, W. (1980). Readability and recall of short prose passages: A theoretical analysis. *Journal of Experimental Psychology: Human Learning and Memory, 6*, 335-354.

Neches, R., & Hayes, J.R. (1978). Progress towards a taxonomy of strategy transformations. In A.M. Lesgold, J.W. Pellegrino, S. Fokkema, & R. Glaser (Eds.), *Cognitive Psychology and Instruction*, New York: Plenum Books.

Neves, D.M., & Anderson, J.R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

Newell, A. (1980). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII.* Hillsdale, NJ: Erlbaum.

Newell, A., & Simon, H.A. (1972). *Human Problem Solving.* New York: Prentice-Hall.

Newell, A., & Rosenbloom, P.S. (1981). Mechanisms of skill and the law of practice. In J.R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum.

Nielsen, J., Mack, R.L., Bergendorff, K.H., & Grischkowsky, N.L. (1986). Integrated software usage in the professional work environments: Evidence from questionnaires and interviews. In M. Mantei & P. Orbeton (Eds.), *Proceedings CHI'86 Human Factors in Computing* (pp. 162-167). New York: Association for Computing Machinery.

Parnas, D.L. (1969). On the use of transition diagrams in the design of a user interface for an interactive computer system. *Proceedings of the 24th National ACM Conference, 15*, 379-385.

Pedhazur, E.J. (1982). *Multiple regression in behavioral research.* New York: Holt, Rinehart, & Winston.

Polson, P.G. (1987). A quantitative theory of human-computer interaction. In J.M. Carroll (Ed.), *Interfacing thought: Cognitive aspects of human-computer interaction.* Cambridge MA: Bradford Books/MIT Press.

Polson, P.G. (1988). The consequences of consistent and inconsistent user interfaces. In R. Guindon (Ed.), *Cognitive Science and Its Applications For Human-Computer Interaction.* Hillsdale, NJ: Erlbaum.

Polson, P.G., Bovair, S., & Kieras, D.E. (1985). Transfer between text editors. In P. Tanner, & J.M. Carroll (Eds.), *Proceedings CHI'87 Human Factors in Computer Systems* (pp. 27-32). New York: Association for Computing Machinery.

Polson, P.G., Kieras, D.E., Engelbeck, G., & Willer, N. (1983). Effects of mental models on the acquisition of operating procedures. *Paper Presented at the Annual Meeting of the Psychonomic Society*, San Diego, CA.

Polson, P.G., & Kieras, D.E. (1984). A formal description of users' knowledge of how to operate a device and user complexity, *Behavior Research Methods, Instruments, & Computers, 16*, 249-255.

Polson, P.G., Muncher, E. & Engelbeck, G. (1986). A test of a common elements theory of transfer. *Proceedings of the CHI'86 Conference on Human Factors in Computing* (pp 78-83), New York: Association for Computing Machinery.

Postman, L. (1971). Transfer, interference, and forgetting. In J.W. King & L.A. Riggs (Eds.), *Woodworth and Schlosberg's experimental psychology*. New York: Holt, Rienhart, and Winston.

Postman, L. & Underwood, B.J. (1973). Critical issues in interference. *Memory and Cognition, 1*, 19-40.

Raaijmakers, J.G.W. & Shiffrin, R.M. (1980). SAM: A theory of probabilistic search of associative memory. In G.H. Bower (Ed.), *The psychology of learning and motivation: Advances in research and theory* (Vol 14, pp. 207-262), New York: Academic Press.

Raaijmakers, J.G.W. & Shiffrin, R.M. (1981). Search of associative memory. *Psychological Review. 88*, 93-134.

Reisner, P. (1981). Formal grammars and human factors design of a graphics system. *IEEE Transactions on Software Engineering, SE-7(2)*, 229-240.

Rosson, M.B. (1983). Patterns of experience in text editing. In A. Janda (Ed.), *Proceedings CHI'83 Human Factors in Computing* (pp 171-175), New York: Association for Computing Machinery.

Rubenstein, R., & Hersh, H.M. (1984). *The human factor: Designing computer systems for people.* Burlington, MA: Digital Press.

Sauers, R., & Farrell, R. (1982). *GRAPES user's manual* (Department of Psychology Technical Report No 1). Pittsburg, PA: Carnegie-Mellon University.

Schneider, W., & Shiffrin, R.M. (1977). Controlled and automatic human information processing: I. Detection, search, and attention. *Psychological Review, 84,* 1-66.

Schwartz, J.P., & Norman, K.L. (1986). The importance of item distinctiveness on performance using a menu selection system. *Behavior and Information Technology, 5(2),* 173-182.

Shields, J.L., Goldberg, S.L., & Dressel, J.D. (1979). *Retention of basic soldiering skills.* Research Report 1225: US Army Research Institute for the Behavior and Social Sciences.

Shneiderman, B. (1982). Multi-party grammars and related features for designing interactive systems. *IEEE Transactions on Systems, Man, and Cybernetics, 12(2),* 148-154.

Shneiderman, B. (1987). *Designing the user interface: Strategies for effective human-computer interaction,* Reading MA: Addison-Wesley Publishing Company.

Sterling, L., & Shapiro, E. (1986). *The art of prolog: Advanced programming techniques.* Cambridge, MA: MIT Press.

Singley, M.K. (1987). *Developing models of skill acquisition in the context of intelligent tutoring systems.* Ph.D. dissertation: Carnegie-Mellon University.

Singley, M.K., & Anderson, J.R. (1985). The transfer of text-editing skill. *International Journal of Man-Machine Studies, 22,* 403-423.

Singley, M.K., & Anderson, J.R. (1988). A keystroke analysis of learning and transfer in text editing, *Human-Computer Interaction, 3,* 223-274.

Smith, D.C., & Mosier, J.M. (1986). *Guidelines for designing user interface software* (EDS-TR-86-278, MTR 10090). Bedford, MA: MITRE Corporation.

Suchman, L.A. (1987). *Plans and situated actions: The problem of human-machine communication.* Cambridge, Great Britain: Cambridge University Press.

Thorndike, E.L., & Woodworth, R.S. (1901). The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review, 8,* 247-261.

Tulving, E., & Pearlstone, Z. (1966). Availability versus accessibility of information in memory for words, *Journal of Verbal Learning and Verbal Behavior, 5,* 381-391.

Waern, Y. (1985). Learning computerized task as related to prior task knowledge. *International Journal of Man-Machine Studies,* 22, 441-455.

Walker, W.H., & Kintsch, W. (1985). Automatic and strategic aspects of knowledge retrieval. *Cognitive Science, 9,* 261-283.

Wasserman, A.I., & Stinson, S.K. (1979). A specification method for interactive information systems. *Proceedings, Specification of Reliable Software, IEEE Computer Society* (pp. 68-79).

APPENDIX A
LISTING OF THE CODE FOR THE MUDDLE SIMULATION

This appendix contains the complete listing of the Muddle simulation. The Muddle simulation was written in LPA MacPROLOG on the Apple Macintosh. Those unfamiliar with PROLOG should consult Sterling and Shapiro (1986) for an introduction to programming in PROLOG. The text that appears in italics within the code are comments.

The code is divided into seven sections: 1) The code for the mac-interface used to run the Muddle simulation; 2) The code implementing the processing structure of Muddle; 3) The code implementing Muddle's retrieval mechanism; 4) Code for strengthening and adding cues, images, and associations to memory; 5) The code used to recognize primitive types; 6) Code implementing Muddle's interface with the application simulation; and 7) Code for input/output and various utility routines needed by the program.

Interface to the Muddle Simulation

This section contains the modeler's interface to the Muddle simulation. This code uses code specific to LPA MacPROLOG and implements pull-down menus and dialog boxes. The clause that is most important in this section of code is *'Task'('Perform task')*. This clause reads in the task description, initializes values for a task, and starts the Muddle simulation. The other clauses are used to load, clear, and display memory and application state information.

```
/* Code installing the pull-down menus. */
'<LOAD>'(_) :-
        install_menu('Task',['Perform task','(-','Show task description','Show solution path','Show trys']),
        install_menu('Memory',['Show memory','Clear memory','(-','Load memory','Save memory']),
        install_menu('System',['Show menu state','(-','Kill menu description', 'Kill task description']),
        wfront('Default Output Window'),
        wsize('Default Output Window', 234, 2, 102, 506),
        'Fonts'('9'),
        showMenuState.

/* Code for reading in the task and menu descriptions, initializing data, and calling the
simulation.*/
'Task'('Perform task') :-
        not idef(association),
        errormessage(['There is nothing in associative memory.']).

'Task'('Perform task') :-
        old('TEXT',File,_),
        kill(task),
        kill(correctSolution),
        remember(trys,1),
        loadTaskFile(File),
        recall(numberOfCues,StartingCues),
        recall(numberOfImages,StartingImages),
        numberOfAssociations(StartingAssociations),
        context(Context),
        on(menuTitle(MT),Context),
```

```
        remember(probe,0),
        remember(menuTitle,MT),
        remember(oldContext,Context),
        remember(salientChanges,Context), !,
        solveProblem(finished(instructions)),
        giveSummary(StartingCues,StartingImages,StartingAssociations).

/* Display the remaining solution path for a task. */
'Task'('Show solution path') :-
        recall(solutionPath,SP),
        write('Solution path: '), write(SP), nl.

/* Display the current task description. */
'Task'('Show task description') :-
        not idef(task),
        message(['No task description currently exists.']).

'Task'('Show task description') :-
            db(task).

/* Display how many times the simulation has tried to complete a task step. */
'Task'('Show trys') :-
        recall(trys,T),
        write('Trys: '), write(T), nl.

/* Display the contents of the associative memory. */
'Memory'('Show memory') :-
        not idef(association),
        message(['You do not have anything in memory that ~M can be shown.']).

'Memory'('Show memory') :-
        nl,
        db(cue),
        db(image),
        db(association).

/* Clear Muddle's associative memory. */
'Memory'('Clear memory') :-
        kill(cue),
        kill(association),
        kill(image).

/* Initialize Muddle's associative memory from a file. */
'Memory'('Load memory') :-
        old('TEXT',File,Volume),
        open(File,Volume),
        consultFile(File),
        close(File).

/* Save the current state of Muddle's associative memory in a file. */
'Memory'('Save memory') :-
        not idef(association),
        message(['You do not have anything in memory that ~M can be saved.']).

'Memory'('Save memory') :-
        idef(association),
        new(File,Vol,'Memory file name:'),
        create(File,Vol,'TEXT'),
        saveMemory(File),
        close(File).
```

```
/* Display the current menu state in a window. */
'System'('Show menu state') :-
        showMenuState.

/* Kill the description of the current menu state. */
'System'('Kill menu description') :-
        killMenuDescription.

/* Kill the description of the current task. */
'System'('Kill task description') :-
        kill(task).

/* Clauses for saving the contents of  Muddle's associative memory in a file. */
saveMemory(File) :-
        cue(CueIndex,Cue,CueVars),
        write(File,cue(CueIndex,Cue,CueVars)), write(File,'.'), nl(File),
        fail.

saveMemory(File) :-
        image(ImageIndex,Image,ImageVars),
        write(File,image(ImageIndex,Image,ImageVars)), write(File,'.'), nl(File),
        fail.

saveMemory(File) :-
        association(CueIndex,ImageIndex,N),
        write(File,association(CueIndex,ImageIndex,N)), write(File,'.'), nl(File),
        fail.

saveMemory(File) :-
        recall(numberOfCues,Cues),
        write(File,'execute(remember(numberOfCues,'),
        write(File,Cues), write(File,')).'), nl(File),
        recall(numberOfImages,Images),
        write(File,'execute(remember(numberOfImages,'), write(File,Images),
        write(File,')).').

/* Display Muddle's predictors in a window . */
giveSummary(StartingCues,StartingImages,StartingAssociations) :-
        recall(probe,Probe),
        recall(numberOfCues,NCues),
        AddedCues is NCues - StartingCues,
        recall(numberOfImages,NImages),
        AddedImages is NImages - StartingImages,
        numberOfAssociations(NAssociations),
        AddedAssociations is NAssociations - StartingAssociations,
        write('Probes : '), write(Probe), tab(2),
        write('Added cues: '), write(AddedCues), tab(2),
        write('Added images: '), write(AddedImages), tab(2),
        write('Added associations: '), write(AddedAssociations), nl,
        beep(100).

numberOfAssociations(Number) :-
        findall(X,association(_,_,X),NX),
        length(NX,Number).
```

Code Implementing Muddle's Processing Structure

This section of code implements Muddle's processing structure. It is losely based on the PROLOG meta-interpreter given in Sterling and Shapiro (1986, page 304).

```
/* Top-level of the Muddle interpreter.
        The first clause halts if it has solved a task.
        The second clause attempts to retrieve a method given a probe set.
        The third clause expands the probe set if a solution is not found. */
solveProblem(_) :-
        recall(solutionPath,[]).

solveProblem(Goal) :-
        recall(salientChanges,SalientChanges),
        retrieve([Goal|SalientChanges],OrderedImageStrengthPairs,Subgoal),
        incrementProbeCount,
        tab(10),write(OrderedImageStrengthPairs), nl,
        solveFor(Subgoal),
        evaluateProblemSolution(Goal,SalientChanges,Subgoal,NewGoal), !,
        solveProblem(NewGoal).

solveProblem(Goal) :-
        context(Context),
        remember(oldContext,Context),
        remember(salientChanges,Context), !,
        solveProblem(Goal).

/* Solve for a retrieved method.
        First clause: aborts an attempt at a solution if an error message is displayed by the application.
        Second clause: breaks two goals into subgoals that are then solved in turn.
        Third clause: checks the application description to see if goal is to check the application state.
        Fourth clause: Checks to see if the goal corresponds to an operator, if it does the operator is performed.
        Fifth clause: Solves for a goal by retrieving subgoals associated with the goal and solving for those
            subgoals. */
solveFor(_) :-
        def(interupt).

solveFor((Goal1,Goal2)) :-
        !,
        solveFor(Goal1),
        solveFor(Goal2).

solveFor(Condition) :-
        condition(Condition).

solveFor(Operator) :-
        operator(Operator), !.

solveFor(Goal) :-
        !,
        not isaCondition(Goal),
        retrieve([Goal],_,Subgoal),
        solveFor(Subgoal),
        evaluateAction(Goal,Subgoal).

/* Evaluates the results of operators.
        First clause: Interrupts a solution if the application displays a message directing an action.
        Second clause: Interrupts a solution if the application displays an error message.
        Third clause: Stores the goal and the subgoals in memory as associations . */
```

```
evaluateAction(Goal,_) :-
        menuDescription(msg((_,do(X)))),
        assert(interupt).

evaluateAction(Goal,_) :-
        menuDescription(msg(made(error))).

evaluateAction(Goal,Subgoal) :-
        store([Goal],Subgoal).

/* Evaluate the change in the state of the application to decide if a method was successful.
        First clause: The method was not successful if the CAI package intervened.
        Second clause: The method was not successful if the application displayed an error message.
        Third clause: The method was not successful if the application displayed an error message.
        Fourth clause: The method was not successful if the application displayed anmessage requesting a
            different action be performed.
        Fifth clause: The method was successful if no error messages were displayed and the CAI package did not
            intervened. */

evaluateProblemSolution(Goal,SalientChanges,_,finished(Solution)) :-
        idef(correctSolution), correctSolution(Solution),
        kill(correctSolution),
        kill(interupt),
        explainAndSolveForSolution(Solution,Explaination),
        store([Goal|SalientChanges],Explaination),
        salientChanges, !.

evaluateProblemSolution(Goal,SalientChanges,_,Goal) :-
        menuDescription(msg(made(error))),
        not on(msg(made(error)),SalientChanges),
        recall(oldContext,OldContext),
        remember(oldContext,[msg(made(error))|OldContext]),
        remember(salientChanges,[msg(made(error))|SalientChanges]),
        kill(interupt), !.

evaluateProblemSolution(Goal,SalientChanges,_,Goal) :-
        menuDescription(msg(made(error))),
        kill(interupt), !.

evaluateProblemSolution(Goal,SalientChanges,_,Goal) :-
        menuDescription(msg((State,do(X)))),
        recall(oldContext,OldContext),
        remember(oldContext,[msg((State,do(X)))|OldContext]),
        remember(salientChanges,[do(X)|SalientChanges]),
        kill(interupt), !.

evaluateProblemSolution(Goal,SalientChanges,Action,finished(Action)) :-
        store([Goal|SalientChanges],Action),
        kill(interupt),
        salientChanges, !.

/* Solve for a solution given to Muddle by the CAI package.
        First clause: A solution can be explained if the methods in memory can be used to derive the given
            solution.
        Second clause: A solution is solved for if it cannot be derived by methods in memory. */
explainAndSolveForSolution((type(Something),press(OtherThing)),(type(Type),press(OtherThing))) :-
        (menuDescription(Type(Something)); task(Type(Something))),
        solveFor((choose(Type(Something)),typeString(Something),
        press(OtherThing))), !.

explainAndSolveForSolution(Solution,Solution) :-
        solveFor(Solution).
```

```
/* Code that selects the salient cues from the description of the application state. */
salientChanges :-
        context(Context),
        rememberSalientChanges(Context).

/* Code for selecting the salient cues from the description of the application state.
        First clause: If the menu changed then select all the unique descriptions of objects.
        Second clause: If the menu did not change, select all unique object descriptions that were not present in
            the previous state description.
        Third clause: If the deletion of message was the only state change delete those messages from the cue
            list. */
rememberSalientChanges(Context) :-
        recall(menuTitle,Title),
        on(menuTitle(MT),Context),
        MT \= Title,
        remember(menuTitle,MT),
        remember(oldContext,Context),
        remember(salientChanges,Context).

rememberSalientChanges(Context) :-
        recall(oldContext,OldContext),
        noteChanges(Context,OldContext,SalientChanges),
        SalientChanges \= [],
        remember(oldContext,Context),
        remember(salientChanges,SalientChanges).

rememberSalientChanges(Context) :-
        recall(oldContext,OldContext),
        removeMsgs(Context,SalientChanges),
        remember(salientChanges,SalientChanges).

noteChanges([],_,[]) :- !.

noteChanges([Cue|Context],PreviousContext,Changes) :-
        on(Cue,PreviousContext), !,
        noteChanges(Context,PreviousContext,Changes).

noteChanges([Cue|Context],PreviousContext,[Cue|Changes]) :-
        !,
        noteChanges(Context,PreviousContext,Changes).

removeMsgs(Context,SalientChanges) :-
        remove(msg(_),Context,Changes), !,
        removeMsgs(Changes,SalientChanges).

removeMsgs(SalientChanges,SalientChanges) :- !.

/* Increment count of how many times memory has been probed. */
incrementProbeCount :-
        recall(probe,P),
        P1 is P ++ 1,
        remember(probe,P1).
```

Code Implementing Muddle's Memory Retrieval Mechanisms

This section of code implements Muddle's memory retrieval mechanisms. Muddle's retrieval mechanism uses the sampling equation of Raaijmakers and Shiffrin (1980; 1981) to search memory. The highest-level clause in this section of code in *retrieve*. This clause is called by the *SolveProblem* and the *SolveFor* clauses given in the previous section. These pass a list of cues to retrieve which passes back a list of ordered image-strength pairs and the strongest image retrieve from memory.

```
/* top-level clause for the retrieval code. */
retrieve(Cues,OrderedImageStrengthPairs,Image) :-
        releventCues(Cues,ReleventCues),
        associatedImages(ReleventCues,[],AssociatedImages),
        AssociatedImages \= [],
        samplingStrengths(AssociatedImages,ReleventCues,ImageStrengthPairs),
        sortPairs(ImageStrengthPairs,OrderedImageStrengthPairs), !,
        on(Strength-ImageIndex,OrderedImageStrengthPairs),
        unifyCuesAndImage(ReleventCues,ImageIndex,Image).

/* A cue is a relevent cue if it is associatied with any image in memory. */
releventCues([],[]) :- !.

releventCues([Cue|Cues],[Cue|ReleventCues]) :-
        matchCue(_,Cue), !,
        releventCues(Cues,ReleventCues).

releventCues([_|Cues],ReleventCues) :-
        !,
        releventCues(Cues,ReleventCues).

/* An image is an associated image if it is associated with a cue on the cue list. */
associatedImages([],AssociatedImages,AssociatedImages) :- !.

associatedImages([Cue|Cues],FoundImages,AssociatedImages) :-
        matchCue(CueIndex,Cue),
        association(CueIndex,ImageIndex,_),
        not on(ImageIndex,FoundImages), !,
        associatedImages([Cue|Cues],[ImageIndex|FoundImages],AssociatedImages).

associatedImages([_|Cues],FoundImages,AssociatedImages) :-
        !,
        associatedImages(Cues,FoundImages,AssociatedImages).

/* The sampling strength of the images are found by finding the strengths for each image and then normalizing
        all the strengths. */
samplingStrengths(Images,Cues,SamplingStrengths) :-
        sampleStrengths(Images,Cues,SSs),
        findall(X,on(X-_,SSs),Strengths),
        map(+,Strengths,0,SumSampleStrengths),
        divideSampleStrengthsBySum(SSs,SumSampleStrengths,SamplingStrengths).

/* The sampling strengths for an image is normalized by dividing by the total
        sampling strengths across images. */
divideSampleStrengthsBySum([],_,[]) :- !.

divideSampleStrengthsBySum([SS-Image|SampleStrengths],SumSampleStrengths,[S-Image|SamplingStrengths]) :-
        S is SS + SumSampleStrengths, !,
        divideSampleStrengthsBySum(SampleStrengths,SumSampleStrengths,SamplingStrengths).
```

```
/* The sampling strength for a list of images is computer by computing the strength for
        each image */
sampleStrengths([],_,[]) :- !.

sampleStrengths([Image|Images],Cues,[SS-Image|SampleStrengths]) :-
        sampleStrengths1(Image,Cues,1,SS), !,
        sampleStrengths(Images,Cues,SampleStrengths).

/* The sample strength of an image is the product of all the associative strengths of that image with the set of
cues. */
sampleStrengths1(_,[],SS,SS) :- !.

sampleStrengths1(Image,[Cue|Cues],CSS,SS) :-
        ss(Cue,Image,S),
        NSS is S * CSS, !,
        sampleStrengths1(Image,Cues,NSS,SS).

/* The sample strength (ss) of an image with a cue is computed using the ss clauses.
        First clause: Computes the strength for an image that if it is associated
            with the cue.
        Second clause: Computes the residual strength for a domain-specific image
            that is not associated with the cue.
        Third clause: Computes the residual strength for a problem-solving image
            that is not associated with the cue. */
ss(Cue,ImageIndex,S) :-
        matchCue(CueIndex,Cue),
        association(CueIndex,ImageIndex,For),
        findall(X,association(CueIndex,_,X),SumList),
        map(++,SumList,0,Sum),
        S is For + Sum, !.

ss(_,ImageIndex,0.01) :-
        integer(ImageIndex), !.

ss(_,ImageIndex,0.09) :-
        not integer(ImageIndex), !.

/* Match cues with index values. */
matchCue(Index,Cue) :-
        cue(Index,GroundCue,Vars),
        tohollow(GroundCue,Cue,Vars).

/* Match images with index values. */
matchImage(Index,Image) :-
        image(Index,GroundImage,Vars),
        tohollow(GroundImage,Image,Vars).

/* Bind the varables in the cues to the retrieved image. */
unifyCuesAndImage(Cues,ImageIndex,UnifiedImage) :-
        image(ImageIndex,Image,ImageVars),
        hollowCuesAndImage(Cues,Image,ImageIndex,
        ImageVars,HollowCues,UnifiedImage),
        unifyCues(HollowCues,Cues).

/* Create variables in the cues and image. */
hollowCuesAndImage(Cues,Image,ImageIndex,ImageVars,HollowCueList,HollowImage) :-
        gatherCuesAndTokens(Cues,ImageIndex,[],CueList,ImageVars,Tokens),
        compactList(Tokens,[],CompactTokens),
        tohollow(CueList-Image,HollowCueList-HollowImage,CompactTokens).
```

```
gatherCuesAndTokens([],_,CueList,CueList,Tokens,Tokens) :- !.

gatherCuesAndTokens([Cue|Cues],ImageIndex,CurrentCueList,CueList,CurrentTokenList,TokenList) :-
        matchCue(CueIndex,Cue),association(CueIndex,ImageIndex,_),
        cue(CueIndex,GroundCue,CueTokens),
        append(CueTokens,CurrentTokenList,NewTokenList), !,
        gatherCuesAndTokens(Cues,ImageIndex,[GroundCue|CurrentCueList],CueList,NewTokenList,TokenList).

gatherCuesAndTokens([_|Cues],ImageIndex,CurrentCueList,CueList,CurrentTokenList,TokenList) :-
        !,
        gatherCuesAndTokens(Cues,ImageIndex,CurrentCueList,CueList,CurrentTokenList,TokenList).

unifyCues([],_) :- !.

unifyCues([H|T],Cues) :-
        on(H,Cues), !,
        unifyCues(T,Cues).

unifyCues([_|T],Cues) :-
        !,
        unifyCues(T,Cues).
```

Code For Storing Cues, Images, and Associations In Memory

This section contains the code for creating, strengthening and storing associations in Muddle's memory.

```
/* Top-level clause for storing cues, images, and associations in Muddle's memory.
        First variables are extracted from the cues and images.
        Next the image is generalized.
        The image is then either strengthened or added to memory.
        Finally, the cues and associations are either strengthen or added to memory. */
store(Cues,Image) :-
        toground(Cues-Image,_,[]),
        variableMappings(Cues,Image,Mappings),
        generalizeImage(Image,Mappings,GeneralImage,ImageTokens),
        addImage(GeneralImage,ImageTokens,ImageIndex,Vars),
        addCuesAndAssociations(Cues,ImageIndex,Mappings).

/* The terms that can be variablized are extracted from the list. */
variableMappings(Cues,Image,Mappings) :-
        cueTokens(Cues,[],CueTokens),
        findImageTokens(Image,[],ImageTokens),
        append(CueTokens,ImageTokens,Tokens),
        compactList(Tokens,[],CompactTokenList),
        tohollow(Cues-Image,_,CompactTokenList,Vars),
        bind(Vars),
        mappingList(CompactTokenList,Vars,Mappings).

mappingList([],[],[]) :- !.

mappingList([HA|TA],[HB|TB],[[HA,HB]|Mappings]) :-
        !,
        mappingList(TA,TB,Mappings).
```

```
/* Add or strengthen cues and associations in memory. */
addCuesAndAssociations([],_,_) :- !.

addCuesAndAssociations([Cue|Cues],ImageIndex,Mappings) :-
        generalizeCue(Cue,Mappings,GeneralCue,CueTokens),
        addCue(GeneralCue,CueTokens,CueIndex),
        addAssociation(CueIndex,ImageIndex), !,
        addCuesAndAssociations(Cues,ImageIndex,Mappings).

/*  Adds or strengthens an image in memory.
            First clause: Adds image to and variable if the image already in memory.
            Second clause: Adds a image to memory if it does not already exist. */
addImage(Image,ImageTokens,ImageIndex,Vars) :-
        tohollow(Image,GeneralImage,ImageTokens),
        matchImage(ImageIndex,ImageX),
        integer(ImageIndex),
        same(ImageX,GeneralImage),
        clausex(image(ImageIndex,StoredImage,Vars),[],1,Position),
        retractx(image,Position),
        assertx(image(ImageIndex,StoredImage,Vars),Position).

addImage(Image,ImageTokens,ImageIndex,ImageTokens) :-
        recall(numberOfImages,NI),
        ImageIndex is NI ++ 1,
        remember(numberOfImages,ImageIndex),
        assert(image(ImageIndex,Image,ImageTokens)).

/*  Adds or strengthens an cue in memory.
            First clause: Adds cue to and variable if the image already in memory.
            Second clause: Adds a cue to memory if it does not already exist. */
addCue(Cue,CueTokens,CueIndex) :-
        tohollow(Cue,GeneralCue,CueTokens),
        matchCue(CueIndex,CueX),
        integer(CueIndex),
        same(CueX,GeneralCue),
        clausex(cue(CueIndex,StoredCue,Vars),[],1,Position),
        retractx(cue,Position),
        assertx(cue(CueIndex,StoredCue,Vars),Position).

addCue(Cue,CueTokens,CueIndex) :-
        recall(numberOfCues,N),
        CueIndex is N ++ 1,
        remember(numberOfCues,CueIndex),
        assert(cue(CueIndex,Cue,CueTokens)).

/*  Adds or strengthens an association in memory.
            First clause: Strengthens an association in memory.
            Second clause: Adds an association to memory if it does not already exist. */
addAssociation(CueIndex,ImageIndex) :-
        clausex(association(CueIndex,ImageIndex,S),[],1,Position),
        retractx(association,Position),
        S1 is S ++ 1,
        assertx(association(CueIndex,ImageIndex,S1),Position).

addAssociation(CueIndex,ImageIndex) :-
        assert(association(CueIndex,ImageIndex,1)).
```

```
/* Extracts tokens that can be variablized in cues. */
cueTokens([],Tokens,Tokens) :- !.

cueTokens([Cue|Cues],CurrentTokens,Tokens) :-
        findCueTokens(Cue,CueTokens),
        append(CueTokens,CurrentTokens,NewTokens), !,
        cueTokens(Cues,NewTokens,Tokens).

generalizeCue(Cue,Mappings,GeneralCue,CueTokens) :-
        findCueTokens(Cue,CT),
        maps(CT,Mappings,MT),
        tohollow(Cue,GeneralCue,CT,CueTokens),
        CueTokens = MT.

findCueTokens(A(_),[]) :-
        (A = press; A = type; A = finished).

findCueTokens(A(B,C),[]).

findCueTokens(msg(A),Tokens) :-
        findCueTokens(A,Tokens).

findCueTokens(A(B),[B]) :-
        (menuDescription(A(_)); task(A(_))),
        atomic(B).

findCueTokens(A(B(C)),Flat) :-
        (menuDescription(B(_)); task(B(_))),
        flattenClause(C,[],Flat).

findCueTokens(A(B(C)),Flat) :-
        not (menuDescription(B(_)); task(B(_))),
        flattenClause(B(C),[],Flat).

findCueTokens(_,[]).

/* Extracts tokens that can be variablized in images. */
generalizeImage(Image,Mappings,GeneralImage,ImageTokens) :-
        findImageTokens(Image,[],IT),
        maps(IT,Mappings,MT),
        tohollow(Image,GeneralImage,IT,ImageTokens),
        ImageTokens = MT.

findImageTokens(A,Tokens,Tokens) :-
        atomic(A).

findImageTokens(A(B),FoundTokens,Tokens) :-
        (menuDescription(A(_)); task(A(_))),
        flattenClause(B,FoundTokens,Tokens).

findImageTokens(A(B),FoundTokens,Tokens) :-
        !,
        findImageTokens(B,FoundTokens,Tokens).

findImageTokens(A,FoundTokens,Tokens) :-
        arity(A,Arity),
        findImageTokens1(Arity,A,FoundTokens,Tokens).

findImageTokens1(0,_,Tokens,Tokens).
```

```
findImageTokens1(N,A,FoundTokens,Tokens) :-
        arg(N,A,Arg),
        findImageTokens(Arg,FoundTokens,NewFoundTokens),
        N1 is N - 1, !,
        findImageTokens1(N1,A,NewFoundTokens,Tokens).

/* Two clauses are the same if they have the same structure and contain the same
        terms. */
same(A,B) :-
        var(A),
        var(B), !.

same((A1,A2),(B1,B2)) :-
        !,
        same(A1,B1),
        same(A2,B2).

same(A1(A2),B1(B2)) :-
        !,
        same(A1,B1),
        same(A2,B2).

same(A,B) :-
        !,
        nonvar(A),
        nonvar(B),
        A = B.

bind([]) :- !.

bind([H|T]) :-
        gensym(var,Var),
        H = Var, !,
        bind(T).

maps([],_,[]) :- !.

maps([H|T],Mappings,[M|Maps]) :-
        on([H,M],Mappings), !,
        maps(T,Mappings,Maps).
```

Code For Muddle's Type Primitives

The section of code given below contains the code used by Muddle to check states of the system.

```
/* Checks to see if  clause "A" is a condition.
        First clause: "A" is a condition if it is part of the menu description.
        Second clause: "A" is a conditon if it is part of the task description.
        Third clause: "A" is a condition if it is a PROLOG primitive.
condition(A) :-
        idef(menuDescription),
        menuDescription(A).

condition(A) :-
        task(A).

condition(A(B)) :-
        def(A),   !,
        A(B).
```

```
isaCondition(highlighted(_)) :- !.

isaCondition(Rel) :-
        menuDescription(Rel(_)), !.

isaCondition(Rel) :-
        task(Rel(_)), !.

instance(X) :-
        not (menuDescription(X(_)); task(X(_))),
        (menuDescription(_(X)); task(_(X))).

/* Function keys used by the menu-system */
isaKey(enter).
isaKey(reqst).
isaKey(cancl).
isaKey(disk).
isaKey(next).
isaKey(exit).
isaKey(ldrt).
isaKey(unldrt).
isaKey(ldlt).
isaKey(unldlt).
```

Code for Interface Between Muddle and Application Simulation

The section of code given below implements the interface between the cognitive processing component of Muddle and the description of the menu state.

```
/* Top-level clause for interpreting Muddle's actions. */
operator(typeString(X)) :-
        functor(X,X,0),
        nonvar(X),
        interpretOperator(typeString(X)), !.

operator(pressKey(X)) :-
        functor(X,X,0),
        nonvar(X),
        interpretOperator(pressKey(X)), !.

/* Clause interprets Muddle's operator.
        First clause: If Muddle attempts to type a string of characters and this step is correct perform the action
            then update the system state.
        Second clause: If Muddle attempts to press a functions key and this step is correct perform the action
            then update the system state.
        Third clause: If Muddle's action is incorrect, and only more than one error has been made then increment
            the count of errors and present the correct action to Muddle.
        Fourth clause: If an error was made increment the error counter. */

interpretOperator(typeString(Action)) :-
        recall(trys,TRYS),
        write(TRYS), tab(2),
        write(Action), tab(1),
        deleteMD(msg),
        checkAction(typeString(Action)),
        doAction(typeString(Action)),
        showMenuState.
```

```
interpretOperator(pressKey(Action)) :-
        recall(trys,TRYS),
        write(Action), tab(1),
        deleteMD(msg),
        checkAction(pressKey(Action)),
        doAction(pressKey(Action)),
        showMenuState,
        remember(trys,1).

interpretOperator(Action) :-
        recall(trys,Trys),
        Trys > 1,
        NTrys is Trys ++ 1,
        remember(trys,NTrys),
        recall(solutionPath,[Solution|SolutionPath]),
        assert(correctSolution(Solution)),
        showMenuState.

interpretOperator(Action) :-
        recall(trys,Trys),
        NT is Trys ++ 1,
        remember(trys,NT),
        showMenuState.

/* Check an action to see if it is correct.  If the action is correct then perform the action. */
checkAction(pressKey(enter)) :-
        not menuDescription(onPromptLine(_)),
        recall(solutionPath,[press(enter)|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(pressKey(enter)) :-
        menuDescription(onPromptLine(X)),
        recall(solutionPath,[(type(X),press(enter))|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(pressKey(enter)) :-
        menuDescription(mainPrompt((press(next),press(enter)))),
        recall(solutionPath,[press(enter)|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(pressKey(X)) :-
        X \= enter,
        recall(solutionPath,[press(X)|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(pressKey(reqst)) :-
        recall(solutionPath,[press(reqst)|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(pressKey(next)) :-
        menuDescription(onPromptLine(X)),
        recall(solutionPath,[press(next)|SolutionPath]),
        remember(solutionPath,SolutionPath),
        nl.

checkAction(typeString(Action)).
```

```
checkAction(pressKey(enter)) :-
        menuDescription(onPromptLine(_)),
        deleteMD(onPromptLine),
        addMD(msg(made(error))),
        assert(interupt),
        !, fail.

checkAction(pressKey(_)) :-
        menuDescription(msg(made(error))),
        !, fail.

checkAction(pressKey(_)) :-
        addMD(msg(made(error))),
        assert(interupt),
        !, fail.

/* Delete a clause/s in the menu description. */
deleteMD(A) :-
        'TUP'(A),
        retract(menuDescription(A)), !.

deleteMD(A) :-
        retractall(menuDescription(A(_))), !.

deleteMD(A) :-
        retractall(menuDescription(A(_,_))), !.

deleteMD(_) :- !.

/* Add a clause to the menu description. */
addMD(Description) :-
        assert(menuDescription(Description)).
```

Code for Input/Output Routines, and Utility Routines

The code presented in this section for I/O routines and various utility routines.

```
/* Opens and reads in a text file from disk. */
consultFile(File) :-
        findfile(File,Vol),
        open(File,Vol),
        readAndAssert(File),
        close(File).

/* Reads in an asserts the clauses contained in a file of PROLOG clauses. */
readAndAssert(File) :-
        read(File,Clause),
        not Clause == end_of_file,
        addClause(Clause), !,
        readAndAssert(File).

readAndAssert(_).

/* Adds execute a clause or add it to PROLOG base of clause . */
addClause(execute(Clause)) :-
        Clause.

addClause(Clause) :-
        assert(Clause).
```

```
/* Load a description of a menu from a file. */
loadMenuFile(File) :-
        findfile(File,Vol),
        open(File,Vol),
        readInMenu(File),
        close(File).

readInMenu(File) :-
        read(File,Clause),
        not Clause == end_of_file,
        addMenuClause(Clause), !,
        readInMenu(File).

readInMenu(_).

/* Add menu clauses to PROLOG base of clauses. */
addMenuClause(execute(Clause)) :-
        Clause.

addMenuClause((Head :- Body)) :-
        assert((Head :- Body)).

addMenuClause(doAction(Action)) :-
        assert(doAction(Action)).

addMenuClause(unhighlighted(_)) :-
        write(unhighlighted), nl.

addMenuClause(A) :-
        addMD(A).

/* Load a description of a menu task from a file. */
loadTaskFile(File) :-
        findfile(File,Vol),
        open(File,Vol),
        readInTask(File),
        close(File).

readInTask(File) :-
        read(File,Clause),
        not Clause == end_of_file,
        addTaskClause(Clause), !,
        readInTask(File).

readInTask(_).

/* Add task clause to PROLOG clause base. */
addTaskClause(execute(X)) :-
        X.

addTaskClause(X) :-
        assert(task(X)).

/* Context gathers all unique descriptions of objects appearing on the menu. */
context(MenuDescription) :-
        makeMenuDescription([],MenuDescription).

makeMenuDescription(CurrentDescription,MenuDescription) :-
        menuDescription(A(B)),
        A \= menuType,
        not (menuDescription(A(C)), B\= C),
        not on(A(B),CurrentDescription), !,
        makeMenuDescription([A(B)|CurrentDescription],MenuDescription).
```

```
makeMenuDescription(MenuDescription,MenuDescription).

/* Code to change an existing menu description and load a new menu description. */
loadMenu(Menu) :-
        killMenuDescription,
        remember(salientChanges,[]),
        banner(loadMenuFile(Menu),['Loading menu: ', Menu]),
        showMenuState.

killMenuDescription :-
        kill(menuDescription),
        kill(doAction).

/* Display the current state of the menu system in a window. */
showMenuState :-
        windows(disp,WL),
        not on('Menu state',WL),
        wcreate('Menu state',1,38,3,179,504),
        wsize('Default Output Window',239,3,98,506),
        wfront('Menu state'),
        'Fonts'('Geneva'),
        'Fonts'('9'), !,
        showMenuState.

showMenuState :-
        recall(solutionPath,[]),
        wfront('Menu state'),
        cursor('Menu state',0,-1),
        clear('Menu state'),
        task(Task),
        write('Menu state','Finished '),
        write('Menu state',Task),
        write('Menu state',' task.').

showMenuState :-
        idef(menuDescription),
        wfront('Menu state'),
        cursor('Menu state',0,-1),
        cut('Menu state'),
        writeTitle('Menu state'),
        writeIds('Menu state'),
        writeSecondaryPrompt('Menu state'),
        writeMainPrompt('Menu state'),
        writeMsgs('Menu state').

writeTitle(_) :-
        not menuDescription(menuTitle(_)).

writeTitle(Window) :-
        menuDescription(menuTitle(MenuTitle)),
        write(Window,MenuTitle), nl(Window), nl(Window).

writeIds(_) :-
        not menuDescription(id(_)).

writeIds(Window) :-
        menuDescription(id(ID)),
        writeId(Window,ID),
        writeCorresponds(Window,ID),
        nl(Window),
        fail.

writeIds(Window) :-
        nl(Window).
```

```
writeId(Window,ID) :-
        menuDescription(highlighted(id(ID))),
        write(Window,'['),write(Window,ID),write(Window,'] '), !.

writeId(Window,ID) :-
        tab(Window,1),write(Window,ID),tab(Window,2), !.

writeCorresponds(Window,ID) :-
        menuDescription(corresponds(id(ID),item(Item))),
        write(Window,Item), tab(Window,1),
        fail.

writeCorresponds(_,_).

writeSecondaryPrompt(_) :-
        not menuDescription(secondaryPrompt(_)).

writeSecondaryPrompt(Window) :-
        menuDescription(secondaryPrompt(SP)),
        write(Window,SP), nl(Window).

writeMainPrompt(_) :-
        not menuDescription(mainPrompt(_)).

writeMainPrompt(Window) :-
        menuDescription(mainPrompt(MP)),
        write(Window,MP), write(Window,' : '),
        writeOnPromptLine(Window).

writeOnPromptLine(Window) :-
        menuDescription(onPromptLine(OPL)),
        write(Window,OPL), nl(Window).

writeOnPromptLine(Window) :-
        nl(Window).

writeMsgs(_) :-
        not menuDescription(msg(_)).

writeMsgs(Window) :-
        menuDescription(msg(Msg)),
        write(Window,Msg), nl(Window).

/* Code for deleting characters that appear on the menu system's main promptline. */
deleteCharacter :-
        not menuDescription(onPromptLine(_)).

deleteCharacter :-
        menuDescription(onPromptLine(X)),
        stringof(SX,X),
        length(SX,1),
        deleteMD(onPromptLine).

deleteCharacter :-
        menuDescription(onPromptLine(X)),
        stringof(StringList,X),
        removeLastItem(StringList,NewStringList),
        stringof(NewStringList,NewString),
        deleteMD(onPromptLine),
        addMD(onPromptLine(NewString)).
```

```
/* Remove last item on a list. */
removeLastItem([],[]) :- !.

removeLastItem([A],[]) :- !.

removeLastItem([H|T],[H|R]) :-
        !,
        removeLastItem(T,R).

/* Find the number of characters in a string. */
numberOfChars(String,NumberOfChars) :-
        stringof(StringList,String),
        length(StringList,NumberOfChars).

/* Update too-many character message on the menu system. */
updateCharMsg :-
        menuDescription(onPromptLine(String)),
        numberOfChars(String,N),
        N > 1,
        addMD(msg((tooMany(characters),do(or(press(del),press(cancl))))))).

updateCharMsg :-
        deleteMD(msg((tooMany(characters),do(or(press(del),press(cancl))))))).

/* Flatten a PROLOG clause to a list of atoms. */
flattenClause(A,Flat,[A|Flat]) :-
        atomic(A).

flattenClause((A,B),Found,Flat) :-
        !,
        flattenClause(A,Found,FlatA),
        flattenClause(B,FlatA,Flat).

flattenClause(A,Found,Flat) :-
        A =.. [Head|_],
        flattenArgs(A,FA),
        append([Head|FA],Found,Flat).

flattenArgs(Functor,FlatArgs) :-
        toground(Functor,GroundFunctor,Vars),
        arity(Functor,N),
        flattenGroundArgs(N,GroundFunctor,[],GroundFlat),
        tohollow(GroundFlat,FlatArgs,Vars), !.

flattenGroundArgs(0,_,Flats,Flats).

flattenGroundArgs(N,GroundFunctor,FoundFlats,Flats) :-
        arg(N,GroundFunctor,Arg),
        N1 is N - 1,
        flattenClause(Arg,[],FlatArgs),
        append(FlatArgs,FoundFlats,NewFoundFlats), !,
        flattenGroundArgs(N1,GroundFunctor,NewFoundFlats,Flats).

/* Find the arity of a PROLOG term. */
arity(Term,Arity) :-
        'TUPLE'(Term,TupleLength),
        Arity is TupleLength - 1.
```

```
/* Remove multiple occurences of an item on a list. */
compactList([],CompactList,CompactList) :- !.

compactList([H|T],CurrentList,CompactList) :-
        on(H,CurrentList), !,
        compactList(T,CurrentList,CompactList).

compactList([H|T],CurrentList,CompactList) :-
        !,
        compactList(T,[H|CurrentList],CompactList).

/* Sort pairs of  items is ascending order, based on the first item of the pair. */
sortPairs(L,SL) :-
        append(L1,[A1-A2,B1-B2|L2],L),
        A1 < B1,
        append(L1,[B1-B2,A1-A2|L2],NL), !,
        sortPairs(NL,SL).

sortPairs(SL,SL) :- !.

/* Remove all occurrences of "Element" for list "List" to create a "NewList". */
removeall(Element,List,NewList) :-
        hollow(Element,HE),
        remove(HE,List,NL), !,
        removeall(Element,NL,NewList).

removeall(_,NewList,NewList) :- !.

/* Ground and then hollow clause "X" creating a new clause (HX) with variables that
        will not unify with the variables in X. */
hollow(X,HX) :-
        toground(X,GX,V),
        tohollow(GX,HX,V).
```

APPENDIX B

REPRESENTATION OF MUDDLE'S INITIAL PROBLEM-SOLVING KNOWLEDGE AS ASSOCIATIONS

Appendix B presents Muddle's representation of its initial problem-solving knowledge as associations. The knowledge presented in this appendix is described as productions in Table 3-3.

```
/* cue(CueIndex,Cue,Vars)
     CueIndex: Number or letter reference for a cue.
     Cue: The cue.
     Vars: Any variables that appear in the propositional representation of the cue.*/
cue(a,mainPrompt(x),[x]).
cue(b,secondaryPrompt(y),[y]).
cue(c,type(something),[something]).
cue(d,press(k),[k]).
cue(e,choose(something(st)),[something,st]).
cue(f,do(or(c1,c2)),[c1,c2]).

/* image(ImageIndex,Image,Vars).
     ImageIndex: Number or letter reference for a image.
     Image: The image.
     Vars: Any variables that appear in the propositional representation of the cue.*/
image(a,x,[x]).
image(b,y,[y]).
image(c,(choose(something(st)),typeString(st)),[something,st]).
image(d,(instance(something),typeString(something)),[something]).
image(e,(isaKey(k),pressKey(k)),[k]).
image(f,
     (task(goal(v1)),corresponds(something(st),v2(v1))),
     [something,st,v1,v2]).
image(g,
     (task(goal(v1(v2(v3(v4))))),corresponds(something(st),v5(v1(v2(v3))))),
     [something,st,v1,v2,v3,v4,v5]).
image(h,
     (task(goal(v1(v2(v3))))),corresponds(something(st),v4(v1(v2)))),
     [something,st,v1,v2,v3,v4]).
image(i,
     (task(goal(v1(v2(v3(v4))))),corresponds(something(st),v5(v1(v2(v3(v6)))))),
     [something,st,v1,v2,v3,v4,v5,v6]).
image(j,
     (task(goal(v1(v2(v3))))),corresponds(something(st),v4(v1(v2(v5))))),
     [something,st,v1,v2,v3,v4,v5]).
image(k,
     (task(goal(v1(v2)))),corresponds(something(st),v3(v1(v4)))),
     [something,st,v1,v2,v3,v4]).
image(l,
     (corresponds(something(st),v1(v2)),corresponds(v1(v2),v3(empty))),
     [something,st,v1,v2,v3]).
image(m,task(something(st)),[something,st]).
image(n,c1,[c1]).
image(o,c2,[c2]).
```

```
/* association(CueReference,ImageReference,Strength,LastTimeUsed)
     CueReference: Number or letter reference for the associated cue.
     ImageReference: Number or letter reference for the associated image.
     Strength: The number of times the cue and image have been successfully paired..
     LastTimeUsed:  The last time (counted in processing cycles) the cue and image
     were associated.*/
association(a,a,2).
association(b,b,1).
association(c,c,2).
association(c,d,1).
association(d,e,1).
association(e,f,6).
association(e,g,5).
association(e,h,4).
association(e,i,3).
association(e,j,2).
association(e,k,1).
association(e,l,6).
association(e,m,6).
association(f,n,1).
association(f,o,1).

/* Initialize the count of cues and images.  Initialize the internal variable-name var. */
execute(remember(numberOfCues,0)).
execute(remember(numberOfImages,0)).
execute(init_gensym(var)).
```

APPENDIX C

LISTINGS OF THE TASK-SELECTION AND SPELLING-TASKS MENUS

The present appendix presents the listing and the code that interprets Muddle's actions for the Task-Selection and the Spelling-Tasks menus. These two menus are needed to perform the check-spelling example given in Chapter III.

The appendix first presents the Task-Selection menu. This menu is the top-level menu of the menu-system simulated in Experiment 2. Next, the Spelling-Tasks menu is presented. This menu is a low-level menu and is typical of the *prompt-answering* menus used in the two experiments.

The Task-Selection Menu

The present section presents the representation of Task-Selection menu used by the Muddle simulation listed in Appendix A. The description of the Task Selection menu contains two components. The first component is a description of the objects appearing on the menu and their correspondences to other objects on the menu. The second component is code for changing the description of the menu state. This is the code that changes the propositional representation in response to the actions performed by Muddle.

```
/* Description of the objects appearing on the Task-Selection menu. */
menuTitle(taskSelection).

id(a).
id(b).
id(c).
id(d).

item(typingTasks).
item(create(document)).
item(revise(document)).
item(paginate(document)).
item(print(document)).
item(documentTasks).
item(change(name(document))).
item(condense(document)).
item(delete(document)).
item(recover(document)).
item(duplicate(document)).
item(change(defaultFormat(document))).
item(change(alternateFormat(document))).
item(disketteTasks).
item(change(name(diskette))).
item(condense(diskette)).
```

```
item(duplicate(diskette)).
item(erase(diskette)).
item(printIndex(diskette)).
item(change(and(description(printer),description(workstation)))).
item(spellingTasks).
item(check(spelling(document))).
item(load(supplement)).
item(clear(supplement)).

typingTasks(create(document)).
typingTasks(revise(document)).
typingTasks(paginate(document)).
typingTasks(print(document)).

documentTasks(change(name(document))).
documentTasks(condense(document)).
documentTasks(delete(document)).
documentTasks(recover(document)).
documentTasks(duplicate(document)).
documentTasks(change(default(format))).
documentTasks(change(alternate(format))).

disketteTasks(change(name(diskette))).
disketteTasks(condense(diskette)).
disketteTasks(duplicate(diskette)).
disketteTasks(erase(diskette)).
disketteTasks(print(index(diskette))).
disketteTasks(change(and(description(printer),description(workstation)))).

spellingTasks(check(spelling(document))).
spellingTasks(load(supplement)).
spellingTasks(load(supplement)).

mainPrompt((type(id),press(enter))).

/* Correspondences between objects appearing on the menu */
corresponds(id(a),item(typingTasks)).
corresponds(id(a),item(create(document))).
corresponds(id(a),item(revise(document))).
corresponds(id(a),item(paginate(document))).
corresponds(id(a),item(print(document))).
corresponds(id(b),item(documentTasks)).
corresponds(id(b),item(change(name(document)))).
corresponds(id(b),item(condense(document))).
corresponds(id(b),item(delete(document))).
corresponds(id(b),item(recover(document))).
corresponds(id(b),item(duplicate(document))).
corresponds(id(b),item(change(defaultFormat(document)))).
corresponds(id(b),item(change(alternateFormat(document)))).
corresponds(id(c),item(disketteTasks)).
corresponds(id(c),item(change(name(diskette)))).
corresponds(id(c),item(condense(diskette))).
corresponds(id(c),item(duplicate(diskette))).
corresponds(id(c),item(erase(diskette))).
corresponds(id(c),item(printIndex(diskette))).
corresponds(id(c),item(change(and(description(printer),description(workstation))))).
corresponds(id(d),item(spellingTasks)).
```

```
corresponds(id(d),item(check(spelling(document)))).
corresponds(id(d),item(load(supplement))).
corresponds(id(d),item(clear(supplement))).

/* Code which interprets the actions of the Muddle model. */
doAction(pressKey(enter)) :-
    menuDescription(onPromptLine(a)),
    loadMenu(menuTypingTasks).

doAction(pressKey(enter)) :-
    menuDescription(onPromptLine(b)),
    loadMenu(menuDocumentTasks).

doAction(pressKey(enter)) :-
    menuDescription(onPromptLine(c)),
    loadMenu(menuDisketteTasks).

doAction(pressKey(enter)) :-
    menuDescription(onPromptLine(d)),
    loadMenu(menuSpellingTasks).

doAction(pressKey(del)) :-
    deleteMD(onPromptLine(_)).

doAction(pressKey(cancl)) :-
    deleteMD(onPromptLine(_)).

doAction(pressKey(reqst)) :-
    loadMenu(menuRequestTasks).

doAction(typeString(Action)) :-
    addMD(onPromptLine(Action)).
```

The Spelling-Tasks Menu

The present section presents the representation of Spelling-Tasks menu used by the Muddle simulation. Again, the description of the Task Selection menu contains two components: a description of the objects appearing on the menu and code for changing the description of the menu state.

```
/* Description of the objects appearing on the Task-Selection menu. */

menuTitle(spellingTasks).

id(a).
id(b).
id(c).

item(check(spelling(document))).
item(load(supplement)).
item(clear(supplement)).

secondaryPrompt((whenFinished(menu),press(enter))).

mainPrompt((type(id),press(enter))).
```

```
/* Correspondences between objects appearing on the menu. */
corresponds(id(a),item(check(spelling(document)))).
corresponds(id(b),item(load(supplement))).
corresponds(id(c),item(clear(supplement))).

/* Code which interprets the actions of the Muddle model. */
doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(id),press(enter)))),
    menuDescription(onPromptLine(a)),
    deleteMD(onPromptLine),
    deleteMD(mainPrompt),
    addMD(mainPrompt((type(document),press(enter)))),
    addMD(highlighted(id(a))).

doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(document),press(enter)))),
    menuDescription(onPromptLine(_)),
    deleteMD(onPromptLine),
    deleteMD(mainPrompt),
    addMD(mainPrompt((type(diskette),press(enter)))).

doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(diskette),press(enter)))),
    menuDescription(onPromptLine(_)),
    deleteMD(mainPrompt),
    deleteMD(onPromptLine),
    deleteMD(highlighted),
    addMD(msg(did(check(spelling(document))))),
    addMD(mainPrompt((type(id),press(enter)))).


doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(id),press(enter)))),
    menuDescription(onPromptLine(b)),
    deleteMD(onPromptLine),
    deleteMD(mainPrompt),
    addMD(mainPrompt((type(supplement),press(enter)))),
    addMD(highlighted(id(b))).

doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(supplement),press(enter)))),
    menuDescription(onPromptLine(_)),
    deleteMD(onPromptLine),
    deleteMD(mainPrompt),
    addMD(mainPrompt((type(diskette),press(enter)))).

doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(diskette),press(enter)))),
    menuDescription(onPromptLine(_)),
    deleteMD(mainPrompt),
    deleteMD(onPromptLine),
    deleteMD(highlighted),
    addMD(msg(did(load(supplement)))),
    addMD(mainPrompt((type(id),press(enter)))).

doAction(pressKey(enter)) :-
    menuDescription(mainPrompt((type(id),press(enter)))),
    menuDescription(onPromptLine(c)),
    deleteMD(onPromptLine),
    deleteMD(mainPrompt),
    addMD(mainPrompt((type(supplement),press(enter)))),
    addMD(highlighted(id(c))).
```