HUMAN COMPUTER INTERACTION, 1990, Volume 5, pp. 1-48
Copyright © 1990, Lawrence Erlbaum Associates, Inc.

The Acquisition and Performance of Text-Editing Skill: A Cognitive Complexity Analysis

Susan Bovair and David E. Kieras
University of Michigan

Peter G. Polson
University of Colorado

ABSTRACT

Kieras and Polson (1985) proposed an approach for making quantitative predictions on ease of learning and ease of use of a system, based on a production system version of the goals, operators, methods, and selection rules (GOMS) model of Card, Moran, and Newell (1983). This article describes the principles for constructing such models and obtaining predictions of learning and execution time. A production rule model for a simulated text editor is described in detail and is compared to experimental data on learning and performance. The model accounted well for both learning and execution time and for the details of the increase in speed with practice. The relationship between the performance model and the Keystroke-Level Model of Card et al. (1983) is discussed. The results provide strong support for the original proposal that production rule models can make quantitative predictions for both ease of learning and ease of use.

This article is based on Bovair, Kieras, and Polson (1988).

Authors' present addresses: Susan Bovair, Perry Building, 330 Packard, University of Michigan, Ann Arbor, MI 48109; David E.Kieras, Electrical Engineering & Computer Science Department, University of Michigan, Ann Arbor, MI 48109; Peter G. Polson, Institute of Cognitive Science, Campus Box 345, University of Colorado, Boulder, CO 80309.

CONTENTS

1. INTRODUCTION

Because of the widespread use of computers in the modern workplace, it has become increasingly important that computer systems and their software are designed to be both easy to learn and easy to use. One good way to do this is to make the knowledge needed to operate a system as simple as possible; in other words, to minimize the *cognitive complexity* of the system. Kieras and Polson (1985) proposed that, by expressing this knowledge as a production system, the cognitive complexity of a proposed system could be quantitatively evaluated both in terms of the ease of learning and the ease of use. There are two important benefits to be obtained by such an analysis. This first is that it can be performed early in the design process, as soon as the user interface is specified, allowing comparison and evaluation of different designs without the expense of building prototypes. The second benefit is being able to predict the amount of transfer of existing knowledge to new systems, which otherwise involves especially expensive and difficult empirical studies.

Kieras and Polson have shown empirical support for the cognitive complexity framework on text editing and menu systems in a series of papers (Polson, Bovair, & Kieras, 1987; Polson & Kieras, 1985; Polson, Muncher Engelbeck, 1986), as did Kieras and Bovair (1986) on a control panel device. However, none of these papers included a full description of the production system models or the principles by which they should be constructed. In this article, these principles are presented followed by a detailed description of a model for a simulated text editor, and it concludes with a detailed comparison of the model to data. This work is a fully documented and revised version of the preliminary and brief analyses appearing in Polson and Kieras (1985) and also outlined in Polson (1987), in which the predictor variables were obtained informally and were not based on a fully specified cognitive simulation model such as the one that is described here.

The simulation model described and evaluated in this article is used to provide quantitative predictions for the time to learn and the time to execute the methods on a simple text editor. A useful model should be able to predict how difficult it will be to learn a particular procedure as a function of how complex the procedure is and which other procedures have already been learned. The transfer model described in Kieras and Bovair (1986), which predicted procedure learning time on a control panel device, is used in the analyses described in this article to predict the time to learn text-editing procedures in different training orders. In considering execution, there are two major questions. The first is whether the model can predict the time subjects take to perform specific editing tasks; in these analyses the simulation model generated predictions by performing the same editing tasks as the subjects. The second question is whether the model can predict the faster performance that subjects display as they become more practiced in using the editor; the model used a set of "expert" rules generated from novice rules to predict the time to perform the editing tasks after 8 days of practice.

Because this article presents not just experimental results, but also includes a complex cognitive simulation model and the methodology used to construct the model, it is necessarily lengthy and complex. The article is organized as follows: Sections 2 and 3 present the theoretical background. In Section 2 the production system model of the user's knowledge is described, with details of the representation and the notation, and with guidelines for generating such models. Some rules for how the novice representation changes with practice are described, and finally, there is a description of the transfer and performance models. Section 3 presents a description of the text editor used in the experiment and the cognitive complexity model generated for it. This model provides a specific detailed example of the representation and illustrates many of the guidelines. Section 4 presents the data collected to test the theoretical predictions. Two experiments and comparisons with the model are reported; the first experiment addresses transfer and learning, and the second examines performance. The final section, Section 5, contains some conclusions that can be drawn from the experiments and a discussion of using the approach as a design tool.

2. COGNITIVE COMPLEXITY MODELS OF USER KNOWLEDGE

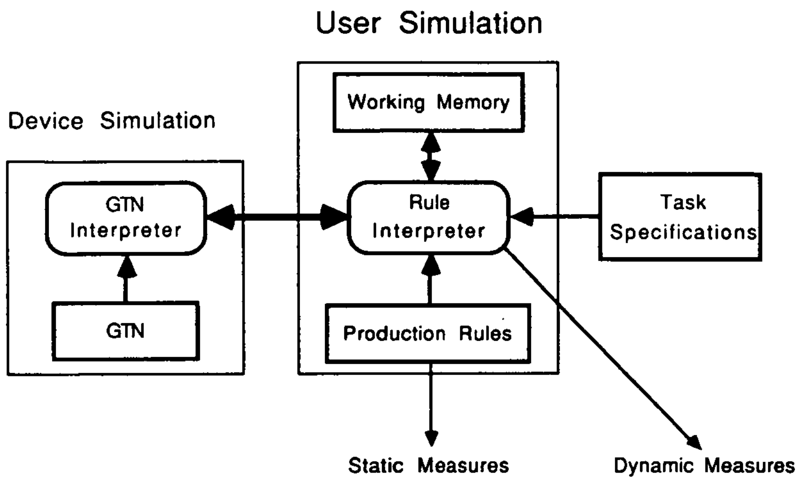
This section describes theoretical background and presents some principles for constructing cognitive complexity models of a user's procedural knowledge. First, the cognitive complexity approach is outlined and compared to other approaches. The production system representation used in cognitive complexity models is then described, followed by the presentation of the principles for constructing such models, and then the derivation of transfer and execution predictions is discussed.

2.1. GOMS and Cognitive Complexity Models

The GOMS model (Card et al., 1983) and the cognitive complexity approach (Kieras & Polson, 1985) both characterize the procedural knowledge the user has to have in order to operate software like an operating system, a text editor, or a database manager. The GOMS formalism describes the *content* and *structure* of this knowledge, whereas the cognitive complexity approach represents the *amount* of the knowledge as well, and, therefore, can be viewed as an extension or elaboration of the GOMS model. In fact, Card et al. (1983), in their description of the Model Human Processor, assume a production system architecture although this is not reflected in their GOMS model.

In the Kieras and Polson (1985) cognitive complexity approach, the predictions of training and performance can be derived from a computer simulation that uses a representation of the user's procedural, or how-to-do-it knowledge, to simulate the user's execution of tasks on a device. Figure 1 shows the components of this simulation. The most important part is the simulation of the user's knowledge formalized as a production system that is described in more detail later. The device model is formalized as a generalized transition network as described in Kieras and Polson (1983, 1985). The purpose of the device model is to generate the correct behavior of the device so that the cognitive complexity model can be shown to generate the correct

Figure 1. **Organization of the user-device interaction simulation used in the modeling work.**



sequences of actions; the details of the device model are not otherwise relevant to the work described here.

Although the cognitive complexity model described in detail in this article is for a text editor, the representation used and the guidelines described can be applied to the construction of any cognitive complexity model. As well as the text editor model described here, production rule models of the cognitive complexity type for interaction with a control panel device (Kieras & Bovair, 1986) and a menu system (Polson et al., 1986) have also been constructed and successfully used to predict difficulty of learning.

The cognitive complexity approach used here has been described elsewhere (Kieras & Polson, 1985; Polson, 1987). It is most closely related to the GOMS model described by Card et al. (1983). Like the GOMS model, cognitive complexity models are descriptions of the knowledge required to use systems and are not intended to be complete simulations of the actual mental processes of the user. According to the rationality principle (Card et al., 1983), when people act to achieve their goals through rational action, it is the task that drives much of their behavior (cf. Newell & Simon, 1972; Simon, 1969). Cognitive complexity models are descriptions of the required knowledge expressed within a constrained production system architecture so that the complexity of the rules and the number of rules needed to express the knowledge are related to the complexity of the system for the user and the amount of knowledge that must be acquired in order to use it.

Production systems are a good choice for building such models because they have already been used successfully as psychological models, particularly in the important and successful work on cognitive skills and their acquisition. One useful feature of production systems is that they demand the construction of explicit and detailed models. They can be constructed according to rules that specify the style in which rules for such models should be written. These *style rules* constrain the production rules to be uniform in size and amount of content; then, as for propositions in text comprehension research, the rules can be counted and used to generate quantitative predictions. Thus, a cognitive complexity model can be viewed as a formalized and quantified GOMS model.

2.2. Comparision With Other Models

Singley and Anderson (1987-1988) used the GRAPES production system to quantify the transfer between two text editors. Their approach was to translate a GOMS model into production rules and then count rules to predict the amount of transfer between editors. Their work is clearly very similar to that described in this article, and it uses a similar approach. One difference is that in this article we attempt to spell out the details of mapping a GOMS model into production rules. In addition, although the transfer models are similar, the cognitive complexity model used here can also predict execution times.

Barnard's interacting subsystems model (Barnard, 1987) is a model human information processor such as that proposed by Card et al. (1983). As such, it is not a model of the user's knowledge like the GOMS model, and it seems to have been used to account for differences found between interfaces rather than to derive quantitative predictions.

The cognitive complexity approach is quite different from the task action grammar (TAG) developed by Payne and Green (Green & Payne, 1984; Payne & Green, 1986) that has been used to analyze computer dialogs in terms of a two-level grammar. Thus, TAG uses the traditional linguistic approach to human-computer interaction. In contrast, the cognitive complexity approach treats human-computer interaction as the acquisition and use of a *cognitive skill.* The TAG approach concentrates on the computer side of the interface, generating a description of the device structure, but without attempting to relate it to the purposes and goals of the user. In developing the cognitive complexity approach, however, the decision was made to separate the structure of the device from the structure of the user's knowledge and to treat them differently (Kieras & Polson, 1985).

2.3. Production System Representation

Production Systems

Production systems, first proposed by Newell and Simon (1972) as a model of the human information processing system, have been used as models of cognitive processes by many workers. Examples include problem-solving

Figure 2. **Example production rules.**

```
(NoviceCopy.P5
   IF  ((GOAL COPY STRING)
        (STEP VERIFY COPY))
THEN  ((VerifyTask COPY)
        (Delete STEP VERIFY COPY)
        Add STEP PRESS ACCEPT)))

(NoviceCopy.P6
   IF  ((GOAL COPY STRING)
        (STEP PRESS ACCEPT))
THEN  ((DoKeystroke ACCEPT)
        (Delete STEP PRESS ACCEPT)
        (Add STEP FINISH UP)))
```

models (Newell & Simon, 1972), models of reading comprehension (Just & Carpenter, 1987; Kieras, 1982), and models of learning and skill acquisition (Anderson, 1983, 1987; Kieras & Bovair, 1986; Singley & Anderson, 1987–1988). The specific production system notation used in this article is the parsimonious production system (PPS) notation described in Covrigaru and Kieras (1987).

A production system consists of a working memory, a collection of production rules, and an interpreter. The working memory contains representations of current goals and inputs from the environment and other information about the state of current and past actions. A production rule is a condition–action pair of the form:

```
IF (condition) THEN (action)
```

The condition of a production rule is a statement about the external environment or the contents of working memory. If the condition of a specific rule is matched, then the rule is said to "fire," and the action of the rule is executed. The interpreter operates by alternating between recognize and act phases. During the recognize phase, the interpreter matches the conditions of all rules against the contents of working memory. During the act phase, all rules that match will fire, and the interpreter will execute their actions. The action part of a rule can consist of several elementary actions; these include the modification of information in working memory by addition or deletion and external actions such as pressing a key.

An example of two production rules in PPS notation can be seen in Figure 2. The first rule has the name NoviceCopy.P5 followed by the rule definition in the form shown earlier. In the example rule, if working memory contained the items (GOAL COPY STRING) and (STEP VERIFY COPY), then the condition of NoviceCopy.P5 would be matched and the rule would fire, executing the action of the rule. Then the actions of verifying the copy, deleting (STEP VERIFY COPY) from working memory and adding (STEP PRESS ACCEPT) to working memory would be performed. A new recognize-act cycle would begin and the condition of NoviceCopy.P5 would then be matched, and this rule would fire.

User's Task Knowledge

The GOMS model (Card, Moran, & Newell, 1980, 1983) describes the knowledge that a user has of a task such as text editing in terms of goals, operators, methods, and selection rules. *Goals* are the representation of a user's intention to perform a task or a part of a task; *operators* are typically the elementary physical or mental actions; *methods* are sequences of operators performed to achieve some goal; and *selection rules* specify the method to be used in a particular context. The user decomposes a complex task into subtasks with a goal defined to perform the task and each subtask.

In the production system formalism, goals can be represented directly, appearing in the conditions of most rules. The goal structure is determined by the relationships between goals appearing in conditions of rules and the actions of other rules that add or remove goals from working memory. Methods can be represented as sequences of rules whose first member is triggered by the appearance of the goal to do the method. Selection rules can be represented as rules triggered by a general goal and a specific context that assert the goal to perform a particular method. Operators consist of elementary actions and more complex actions that test the environment.

In the cognitive complexity approach, the production system model of the user makes no attempt to represent fundamental cognitive processes such as those that comprehend the manuscript in a word-processing task or perceptual processes such as those that determine that the cursor of a text editor is on the correct letter. Kieras and Polson (1985) proposed that dealing with the representation of such processes would greatly increase the complexity of the model of the user, without a corresponding gain in understanding of the cognitive complexity of a particular user interface design. Such complex unanalyzed cognitive processes are represented by the appearance of special operators in the rule actions such as LookMSS, which represents the user's looking in the manuscript for some needed information (see Kieras, 1988 for further discussion). Similar considerations motivate the system's lack of knowledge of the semantics of the items that appear in conditions and actions.

PPS Rule Notation

A PPS production rule consists of five terms enclosed in parentheses:

```
(<name>IF<condition>THEN<action>)
```

The label or name of the rule is not functional but is useful for the programmer. The condition of a rule is a list of clauses that must all be matched for the condition of the rule to be true. Each clause of a condition is a test for the presence of a pattern in working memory. A test for the absence of a pattern in working memory can be made by negating the pattern in the condition with a NOT function. The elements of a pattern may be constants, or variables indicated by the prefix ?. The PPS interpreter decides what values can be assigned to a variable to make the condition true, and these values are used when the action of the rule is executed. If there is more than one possible set of values, then the action is executed once for each set. Variables keep these values only within the scope of the rule in which they appear.

The actions are sequences of operators that can modify working memory to add and delete goals and notes, usually enabling different rules to fire in the next cycle. The actions of a rule can also generate external actions that will be passed to the device simulation and cause it to generate the appropriate response such as changing the screen display. For example, the operator DoKeystroke simulates the action of pressing a single key on an editor keyboard. In addition, actions can be complex operators such as LookMSS, which is a placeholder for the scanning of a marked-up manuscript to find information such as where the next edit is.

The interpreter contains no *conflict resolution* or *data refractoriness.* On any cycle, any rule whose conditions are currently satisfied will fire. This means that rules must usually be written so that a single rule will not fire repeatedly and that only one rule will fire on a cycle. Because these aspects of control structure are not built into the interpreter, the rules must make the flow of control explicit. This was a deliberate decision, intended to ensure that almost all of the procedural knowledge was explicit in the rules, rather than being implicit in the interpreter.

Because the values of variables are computed separately for each recognize-act cycle, the particular values of a variable in one rule are not known in another rule unless the information is explicitly saved in working memory. This ensures that items that the user must retain in short-term memory are explicitly designated (as goals or notes) and are not kept hidden as variable bindings.

2.4. Style Rules for Cognitive Complexity Models

Because production rules are a general programming language, arbitrary programs can be written that generate the correct sequences of user actions. If production systems are to be considered as serious psychological models, however, the structure of the set of rules must be constant, reproducible, and have some claim to psychological content. This can be ensured by *style rules* that constitute assumptions about the representation of procedural knowledge in terms of production rules within the GOMS model framework. A detailed description of a set of style rules follows that constitutes a first step toward a complete set. These rules were generated in an attempt to ensure uniformity of models constructed by different people, but they do not attempt to cover every possible stylistic issue of constructing a production rule representation; some of these can only be addressed empirically. The style rules provide some extra constraint on the model that can be written and make it easier to write. In addition to the style rules, notation conventions are described. These conventions are for convenience only; they simply make rules easier for the transfer simulation to handle.

Notation Conventions

Because the PPS interpreter imposes very few restrictions on the syntax of rules, some conventions on the content of the rules were adopted to produce uniform and consistent rules.

Clause Conventions. In a production rule, condition clauses have the form: (Tag Term_1 Term_2 . . . Term_k). By convention, the number of terms that follow a particular type of tag is fixed: for example, two terms follow the GOAL tag, three follow DEVICE.

Clause Tag Conventions. The tag of a condition clause indicates the type of item that it is, for example, GOAL or NOTE. There are four tags normally used in cognitive complexity analyses: GOAL, NOTE, STEP, and DEVICE. These four seem to be adequate for all the models that have been built so far, with the possible addition of an LTM tag to denote knowledge that the user is assumed to have already. GOAL denotes a goal clause that corresponds directly to the goals used in the GOMS model. Goal clauses are used to trigger selection rules or to perform a particular method. The tag NOTE is used to indicate information kept in working memory over the course of several rule firings. Typically, NOTEs are used to pass parameters and to maintain information from one recognize-act cycle to another. NOTEs can provide specific context for a method and can also be used to control execution ("lockout" notes, to be discussed). The tag STEP is used for the "step goals" that control the execution sequence within a method. The tag DEVICE identifies the information that is provided directly by the device. An example from text editing is the current cursor position, which is visually available on the device screen.

Goal Clause Convention. The format of a goal clause is (GOAL Term_1 Term_2). The convention is that Term_1 is the "verb" or action of the goal, and Term_2 represents the object of the verb or action. Thus, the goal of deleting a character is represented as (GOAL DELETE CHARACTER).

Style Rules

The style rules are presented both as a set of rules (such as those that follow) and as "templates" that can be adapted to specific requirements. The template is a generic set of rules that can be modified by changing some of the terms in the rules from general terms into specific ones. The general terms are shown on the templates enclosed in angle brackets. For example, to generate rules for the top-level unit task structure for an actual analysis of an editor, the template shown in Figure 3 could be modified by replacing <TOP-LEVEL-TASK>with EDIT-MANUSCRIPT, and the new rules could be run.

Rule 1: The production rules must generate the correct sequence of user actions.

This is a minimum requirement for any how-to-do-it representation and should be checked explicitly.

Rule 2: The representation should conform to structured programming principles.

The production rule representation should be written top-down as a strict hierarchy. This means that routines will be called in a standard way, will not affect the caller's context, and will delete local information they use or create.

Rule 3: The top-level representation for most tasks should have a unit task structure.

The unit task structure was proposed by Card et al. (1980, 1983); a task such as editing a manuscript is broken up into a sequence of *unit tasks.* For example, in text editing, each unit task involves searching the manuscript for the next *edit,* discovering what is to be done, and then selecting and executing the appropriate method. After each unit task is completed, the manuscript is checked to see if there are more tasks to be done or not. If there are, the next one is performed, and if there are no more tasks to be done, then the editing session is terminated. This is the "default" top-level structure for a cognitive complexity model, not just of text editing, but for most human-computer interaction situations.

Rule 4: The top-level unit task structure should be based on the top-level goal.

The appearance of the top-level goal should trigger the rule that acquires the next unit task and add the goal of performing it. Figure 3 shows a top-level unit task template.

The first rule starts up the top-level task, while the second acquires the next unit task and adds the goal of performing it, and the last rule recognizes when there are no more tasks to do and stops the system.

Figure 3. **Top-level unit task template.**

```
(Top.Start      IF  ((GOAL PERFORM <TOP-LEVEL-TASK>)
                     (NOT (NOTE PERFORMING <TOP-LEVEL-TASK>)))
              THEN  ((Add STEP GET <UNIT-TASK>)
                     (Add NOTE PERFORMING <TOP-LEVEL-TASK>)))
(Top.P1         IF  ((GOAL PERFORM <TOP-LEVEL-TASK>)
                     (STEP GET <UNIT-TASK>))
              THEN  ((<Get Next Unit Task>)
                     (Add GOAL PERFORM <UNIT-TASK>)
                     (Add STEP CHECK TASKS-DONE)
                     (Delete STEP GET <UNIT-TASK>)))
(Top.Finish     IF  ((GOAL PERFORM <TOP-LEVEL-TASK>)
                     (STEP CHECK TASKS-DONE)
                     (NOTE NO-MORE TASKS))
              THEN  ((Delete GOAL PERFORM <TOP-LEVEL-TASK>)
                     (Delete STEP CHECK TASKS-DONE)
                     (Delete NOTE PERFORMING <TOP-LEVEL-TASK>)
                     (Delete NOTE NO-MORE-TASKS)
                     (Stop Now)))
```

Rule 5: Selection rules select which method to apply.

Selection rules are a key idea in the GOMS model. Given a general goal, they test for specific context and add the goal to perform a specific method. A selection rule consists of two production rules: a Start and a Finish. In the selection rule shown in Figure 4, the Start rule, SelectMethod, fires when the general goal (GOAL PERFORM TASK) appears in working memory with the particular context represented in the NOTEs, and the action asserts the goal of performing a specific method.

The Finish rule, FinishSelect, fires when the note appears that the specific method is complete, removing the general goal, and any locally used NOTEs or STEPs, and adding the note that the general goal has been satisfied. If the same method could be invoked from several different contexts, then a selection rule would be needed for each one of those contexts.

Rule 6: Information needed by a method should be supplied through working memory.

When a method is invoked, there is often information that must be provided for it to operate on. For example, in a text editor, a method for moving the cursor needs to be supplied with the destination position of the cursor. Such information is passed by means of notes in working memory.

Figure 4. **Template for a method.**

```
(SelectMethod   IF ((GOAL PERFORM <TASK>)
                    (NOTE <SPECIFIC CONTEXT>)
                    (NOT (NOTE EXECUTING <TASK>)))
              THEN ((Add GOAL <DO SPECIFIC-METHOD>)
                    (Add NOTE EXECUTING <TASK>)))
(FinishSelect   IF ((GOAL PERFORM <TASK>)
                    (NOTE <SPECIFIC-METHOD> FINISHED))
              THEN ((Add NOTE <TASK> PERFORMED)
                    (Delete GOAL PERFORM <TASK>)
                    (Delete NOTE <SPECIFIC CONTEXT>)))
(StartMethod    IF ((GOAL <DO SPECIFIC-METHOD>)
                    (NOT (NOTE EXECUTING <SPECIFIC-METHOD>))
              THEN ((Add NOTE EXECUTING <SPECIFIC-METHOD>)
                    (Add STEP DO <First-Step>)))
(MethodRule1    IF ((GOAL <DO SPECIFIC-METHOD>)
                    (STEP DO <First-Step>))
              THEN ((<DoAct FirstAct>)
                    (Add STEP DO <Finish-Step>)
                    (Delete STEP DO <First-Step>)))
(FinishMethod   IF ((GOAL <DO SPECIFIC-METHOD>)
                    (STEP DO <Finish-Step>))
              THEN ((Add NOTE <SPECIFIC-METHOD> FINISH)
                    (Delete GOAL <DO SPECIFIC-METHOD>)
                    (Delete STEP DO <Finish-Step>)
                    (Delete NOTE EXECUTING <SPECIFIC-METHOD>)))
```

Rule 7: Sequencing within a method is maintained by chaining STEPs.

While executing a method, the sequence of steps in the routine must be executed in order. This is accomplished by chaining STEPs to maintain the correct sequence. Each method has a Start rule that asserts the first STEP, and the presence of that STEP is tested for by the condition of the next rule in the sequence. When that rule fires, it deletes the current STEP and adds a STEP for the next step in the method. When the sequence is complete, the final rule in the method should delete the goal to do the method and add a NOTE that the method is finished. When using STEPs in this way, all rules except Start and Finish rules will have a single STEP in their condition; a single rule cannot have two STEPs.

In Figure 4, the method template illustrates the chaining of STEPs in a method. The rule StartMethod starts up execution by adding the first STEP. Rule MethodRule1 tests for the presence of (STEP DO <First-Step>) and then performs the action appropriate for the first step. It also deletes the first step clause and adds the next, which in this case is the Finish step.

Rule 8: Labels for STEPs should be based on the action of the rules and should be used consistently.

The labels for STEPs should be chosen subject to two constraints. The first is that in any set of rules that will execute in sequence, the step labels must be chosen in order to preserve the sequential structure. For example, if the STEPs in two rules of a sequence have the same step label, then they may fire at the same time if their other conditions happen to be met.

The second constraint of step labels is that they should be based on the principal action of the rule. Thus, a rule whose action does a VerifyPrompt on a "To where?" prompt would be labeled as (STEP VERIFY-PROMPT TO-WHERE). The label is not just VERIFY-PROMPT because there may be several different prompts to be verified in a single method. Basing the label on the principal action means that the labels will not be arbitrary and that different analysts will generate similar labels for the same STEP. Another approach would be to use simply the number of the step in sequence, but these two approaches cannot be distinguished in any of the models tested so far.

Rule 9: Use a lockout NOTE to prevent a rule from firing repeatedly.

Because a method is triggered simply by the appearance of the goal to do the method, a Start rule will fire as long as that goal remains in working memory due to the lack of data refractoriness in the production rule interpreter. To prevent this from happening, a Start rule adds a note to working memory that the method is in progress. The condition of the Start rule tests for the *absence* of this note with a NOT form clause. Thus, as soon as the Start rule fires, the note will be added and the rule will be locked out from firing again. Because the last rule of a method deletes this lockout note and the goal at the same time, the method will not be executed until the goal is added again.

In Figure 4, use of lockout notes can be seen for both the StartMethod rule and the SelectMethod rule. In StartMethod, the appearance of the goal to do the specific method adds the goal of doing the first step and tests that the lockout note is absent. If this rule fires, it will add the goal of doing the first step and the lockout note so that it cannot fire again. In both of the Finish rules in Figure 4, the lockout note is deleted along with its goal.

Representing the Novice User

The term *novice user* used here does not mean *naive user*. A novice user has acquired all the methods and can execute them correctly but has not had a chance to extensively practice them. A characterization of a novice's knowledge is especially important because it is the basic characterization of what has to be learned and, therefore, is critical to predicting learning time and transfer. Thus, a novice model represents the knowledge that the novice user must acquire in order to operate the system correctly; the model does not attempt to represent the processes by which the knowledge is acquired.

Novice Rule 1: Each overt action requires a separate production rule.

Novice methods are represented so that each overt step in the method is represented in a separate rule. Such steps include not just actions such as pressing keys or flipping switches, but also actions such as looking at an editor prompt on the screen or looking at a manual. Manipulations of working memory such as adding and deleting notes are not considered to be separate steps, and so there may be several of such manipulations in a single rule.

Novice Rule 2: Novices explicitly check all feedback.

It is assumed that novices explicitly check for all feedback from the device before performing the next step in the method. This means that each prompt will be explicitly checked and so represented in a separate production rule.

Representing the Practiced User

Our basic assumption is that it is rules that change with practice, not the speed of execution of the rules. This means that practice in a domain such as text editing will cause changes in the methods, and these changes will be reflected in the number and content of the production rules. Clearly, describing how to transform a novice rule set into an expert rule set is, implicitly, a theory of how expertise develops and how expert and novice procedural knowledge differ. Rather than developing a comprehensive model of the processes involved, we have attempted to devise a simple method for estimating these changes based on previous work on expertise.

Theoretically, there are several alternative ways to represent expertise within the present framework. Card et al. (1983) defined the process of acquiring skill as a change in problem-solving processes, such as the increase in search-control knowledge described by Newell (1980). The representation of a skill can be made more *compact* by using the reduced problem space, which contains fewer states through which to search. Similarly, the mechanism of *composition,* in Anderson's (1987) description of skill acquisition, increases the compactness of the representation by collapsing production rules that always execute in a fixed order into a single rule. The representation can be made more *specialized* by using the skill space, which contains new operators and a minimum of operations. Anderson's *proceduralization* mechanism also produces a more specialized representation.

In our approach, practice is assumed to act by making the novice rule set more compact rather than generating new specialized methods. The main reason for this is that it is simple to define a method for compacting a set of rules based simply on the rule syntax. In contrast, developing specialized methods depends on a detailed analysis or simulation of the system's learning history. We are interested in exploring the value of a simpler analytic approach. Although the rules resulting from compacting the novice rule set are called "expert" in this article, it is important to note that our subjects are not true experts, having had only eight practice sessions. Thus, the expert rule set represents a well-practiced user rather than a true expert.

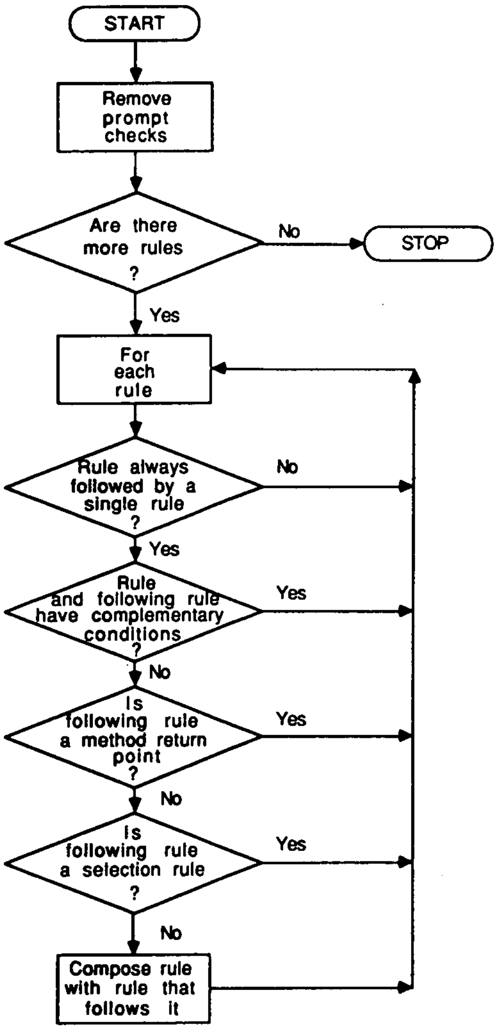
One way to compact methods is to assume that practiced users, unlike novices, do not check every prompt provided by the system. For example, in the text editor used for the experiments described later, the prompt Delete what? appears on the screen after the DELETE key is hit. A novice model includes a rule to check that this prompt appears, whereas the model of the more practiced user does not. However, it is not clear whether or not the more practiced user also verifies that a task has been performed correctly. For example, in the experimental editor, the range of text to be operated on is highlighted before the edit operation is performed. It is possible that even practiced users verify that this highlighting is correct before performing the operation. Because of this uncertainty, we assumed that, in the expert rule set, such task verifications were always performed, and the time estimates resulting from fitting the model to the data could then determine the correctness of this assumption.

Another way we increased compactness is through a mechanism similar to composition (Anderson, 1982, 1987), where rules that always execute in a fixed order are composed into a single rule. In Anderson's model, composition is based on the execution history of the rules; but in this model, composition is done by collapsing rule sequences based on the *content* of the rules. This is possible because the sequencing STEP goals in the rules cause the rules to be executed in a fixed order. The same principle also means that if a method has a selection rule, the selection rule and Start rule of the method will be composed together. More details of this process follow.

Another potential source of increased efficiency for practiced users is that they may have different strategies for performing some of the complex operations such as searching the manuscript in a text-editing task or verifying the correctness of a new state. However, in the model, these operations are ones that have not been represented as rules but simply as complex operators. Any such changes in strategy would be reflected in the estimated time to perform the operator, with less time needed by the more experienced subjects.

In summary, representing how the methods change with experience has been restricted to creating a more compact representation by collapsing rules to remove extra steps and removing prompt-checking steps. The result is an expert rule set with the same goal and method structure and the same general sequence of events within methods as the novice rule set. The expert set is

Figure 5. **Flowchart of expert rule generation process.**


START
Remove prompt checks
Are there more rules ?
No
STOP
Yes
For each rule
Rule always followed by a single rule ?
No
Yes
Rule and following rule have complementary conditions ?
Yes
No
Is following rule a method return point ?
Yes
No
Is following rule a selection rule ?
Yes
No
Compose rule with rule that follows it

considerably shorter than the novice set, which means that all the overt actions in an expert method can be achieved within fewer recognize-act cycles than for novice methods.

Expert Rule Generation Process. The steps of the expert rule generation process are shown in Figure 5. First, any prompt-checking rules are removed. Then, each remaining rule is checked as follows to see if it can be composed with the rule that will follow it in execution: There must always be a single

Figure 6. **Example expert method.**

```
(SelectMethod   IF ((GOAL PERFORM <TASK>)
                    (NOTE <SPECIFIC CONTEXT>)
                    (NOT (NOTE EXECUTING <TASK>)))
              THEN ((Add GOAL <DO SPECIFIC-METHOD>)
                    (Add NOTE EXECUTING <SPECIFIC-METHOD>)
                    (<DoAct FirstAct>)
                    (Add STEP DO <Finish-Step>)))
(FinishMethod   IF ((GOAL PERFORM <TASK>)
                    (GOAL <DO SPECIFIC-METHOD>)
                    (STEP DO <Finish-Step>)
              THEN ((Add NOTE <TASK> PERFORMED)
                    (Delete GOAL PERFORM <TASK>)
                    (Delete GOAL <DO SPECIFIC-METHOD>)
                    (Delete STEP DO <Finish-Step>)
                    (Delete NOTE EXECUTING <SPECIFIC-METHOD>)
                    (Delete NOTE <SPECIFIC CONTEXT>)))
```

rule that executes immediately after the rule in question. This following rule must not contain conditions that are complementary to the first rule. To preserve the control structure of the rules, the following rule must not be a selection rule or method return point. If the following rule meets these conditions it can be composed with the first rule.

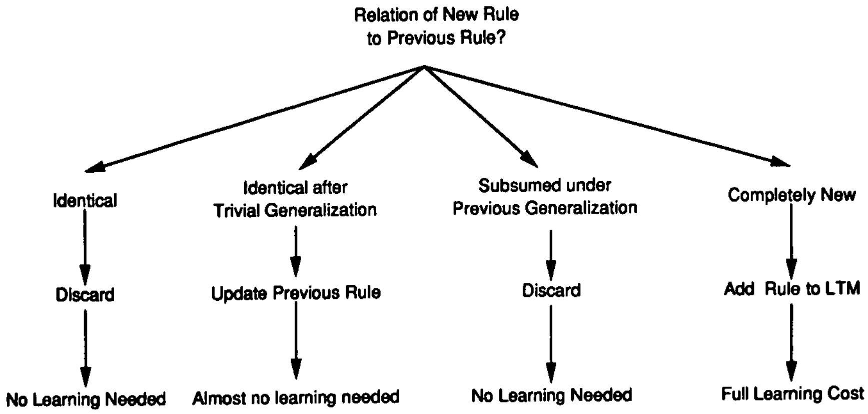
An example of the rule set generated by this process is shown in Figure 6, which shows what the novice method template in Figure 4 would look like for an expert. There are no rules that check prompts. The first step of the original method can be collapsed with the Start rule of the method and that, in turn, with the Start rule from the selection rule. Similar processes operate elsewhere in the method to reduce the five-rule novice method to the two-rule expert method. A further example in the description of the cognitive complexity model for the experimental text editor follows.

2.5. Transfer Model

The original form of the transfer model is described in Kieras and Bovair (1986) and can be considered a modern version of the classical common elements theory of transfer originally proposed by Thorndike and Woodward (1901), in which the common elements are production rules. The transfer process is assumed to be based only on the syntax of rules; no attention is paid to the semantic content.

The transfer process was simulated by a Lisp program based on the one used in Kieras and Bovair (1986). The transfer simulation is given a series of sets of production rules; in this work, each set represents a procedure for one of

Figure 7. **Transfer process flowchart showing possible transfer status of a new rule.**


Relation of New Rule to Previous Rule?
Identical
Identical after Trivial Generalization
Subsumed under Previous Generalization
Completely New
Discard
Update Previous Rule
Discard
Add Rule to LTM
No Learning Needed
Almost no learning needed
No Learning Needed
Full Learning Cost

the editing functions and is based on the content of the training materials used in the experiment for that editing function. The sets are presented to the transfer simulation in the same order as the functions were trained in the experiment. The transfer model allows specific predictions to be made of the amount that must be learned for each of the methods for operating a text editor, making different predictions for the same method depending on what other methods have already been learned. The basic assumption of the transfer model is that the amount that must be learned is linearly related to the time it takes to learn. The production system representation provides a simple way to quantify the amount that must be learned in order to make such predictions. Note that the GOMS model, as described in Card et al. (1983), does not provide a direct quantification. The transfer model also makes the simplifying assumption that the linear relationship holds no matter what cognitive processes are involved in how the procedural knowledge is learned.

Transfer Process

For each new rule set, the simulation considers each rule a candidate for possible transfer with the rules that have already been learned. If it does not transfer, it must be learned, that is, added to the set of known rules. Figure 7 is a flowchart that shows the transfer process and the possible transfer status of a candidate rule; Figure 8 provides some examples drawn from the transfer processing of the experimental editor whose cognitive complexity model is described in Section 3.2.

For each candidate rule, there are four possible outcomes from the transfer process. First, the candidate rule could be *identical* to some existing rule. This means that it does not need to be added to the set of known rules and,

Figure 8. **Examples of rules showing transfer status.**

Example 1: New Rule Is Generalized With the Existing Rule

Old Rule

```
(NoviceDelete.P5
     IF     ((GOAL DELETE STRING)
             (STEP VERIFY DELETE))
   THEN     ((Verify Task DELETE)
             (Delete STEP VERIFY DELETE)
             (Add STEP PRESS ACCEPT)))
```

New Rule:

```
(NoviceCopy.P5
     IF     ((GOAL COPY STRING)
             (STEP VERIFY COPY))
   THEN     ((Verify Task COPY)
             (Delete STEP VERIFY COPY)
             (Add STEP PRESS ACCEPT)))
```

Generalized Rule:

```
(NoviceDelete.P5
     IF   ((GOAL ?X STRING)
           (STEP VERIFY ?X))
   THEN   ((Verify Task ?X)
           (Delete STEP VERIFY ?X)
           (Add STEP PRESS ACCEPT)))
```

Example 2: New Rule Is Subsumed Under an Existing Generalization

Old Rule:

```
(NoviceDelete.P5
     IF     ((GOAL ?X STRING
             (STEP VERIFY ?X))
   THEN     ((VerifyTask ?X)
             (Delete STEP VERIFY ?X)
             (Add STEP PRESS ACCEPT)))
```

New Rule:

```
(NoviceMove.P5
     IF     ((GOAL MOVE STRING)
             (STEP VERIFY MOVE))
   THEN     ((VerifyTask MOVE)
             (Delete STEP VERIFY MOVE)
             (Add STEP PRESS ACCEPT)))
```

(*Continued*)

Figure 8. (*Continued*)

Example 3: New Rule Is Different From All Existing Rules

Old Rule:

```
(NoviceInsert.P2
       IF    ((GOAL INSERT STRING)
              (STEP CHECK-PROMPT INSERT))
              (DEVICE USER MESSAGE Insert What))
    THEN     ((Delete STEP CHECK-PROMPT INSERT)
              (Add STEP LOOKUP MATERIAL))
              (Add STEP PRESS ACCEPT)))
```

New Rule:

```
(NoviceDelete.P2
       IF    ((GOAL DELETE STRING)
              (STEP CHECK-PROMPT DELETE))
              (DEVICE USER MESSAGE Delete What))
    THEN     ((Delete STEP CHECK-PROMPT DELETE)
              (Add GOAL SELECT RANGE)
              (Add STEP SELECT-RANGE DELETE)))
```

therefore, no learning would be needed. Second, the candidate rule could be *generalized* with some existing rule that is similar. The first example in Figure 8 shows two rules that can be generalized as described in more detail later. A generalized rule is created and replaces the original rule, but this is assumed to involve very little learning. Third, the candidate rule could be subsumed under an existing generalized rule as described later, requiring no learning. Example 2 of Figure 8 shows a generalized rule and a rule that can be subsumed under it. Finally, the candidate rule may not fall into any of these three categories, meaning that it is a *new* rule that must be added to the rule set, requiring the full amount of learning. Kieras and Bovair (1986) found that learning time is mostly a function of the number of new rules, although the presence of rules with other transfer status can also increase learning time. In Example 3 of Figure 8, the conditions of the two rules make them apparent candidates for generalization, but, because the actions of the rules are clearly different, the transfer simulation counts NoviceDelete.P2 as NEW.

Generalization

The transfer model performs only a very limited form of generalization, which is adequate to account for the transfer data to which it has been applied, but more powerful forms of generalization in learning may well exist.

Example 1 of Figure 8 shows an example of generalization. The rules are similar but the goal clauses of the two rules differ in the verb of the clause (DELETE in one, COPY in the other). The transfer model forms a generalized rule by replacing the specific verb term with a variable both in the goal clause and wherever it appears in the rule, provided that the difference in verb terms is the only point of difference between the two rules.[1]

If a new rule can be generalized with an existing rule, then the transfer model will count it as GENERALIZED. If the new rule fits the same generalization criterion with an existing rule that has already been generalized, then it will be counted as SUBSUMED. In Example 2 of Figure 8, the rule NoviceDelete.P5 has already been generalized and the only difference between it and NoviceMove.P5 is that NoviceDelete.P5 has a variable form, where NoviceMove.P5 has the specific term MOVE. Thus, the transfer model counts NoviceMove.P5 as SUBSUMED. Thus, if the three methods, DELETE, COPY, and MOVE, are learned in that order, only NoviceDelete.P5 will incur the full learning cost as a new rule, NoviceCopy.P5 will result in a generalized rule, and NoviceMove.P5 will be subsumed.

[1] This generalization criterion is a less restricted form than the one used in Kieras and Bovair (1986). In fact, no differences between the two were observed in these data. See Bovair, Kieras, and Polson (1988).

Independence of Transfer and Execution Models

The number of rules generated by the transfer model using the generalization criterion previously described will be smaller than the full description of the cognitive complexity model. If a different generalization criterion were used, then the resulting rule set would probably be a different size. It is important to note that these differences only affect transfer predictions and, thus, the learning component of the cognitive complexity model; they have no effect on execution. Looking at Figure 8, it is easy to see why this must be so. If the generalized version of NoviceDelete.P5 is in the rule set, then the specific DELETE and COPY versions will not be present. If the generalized version is not in the rule set, then the specific versions will be present. In executing the rules, it does not matter whether the generalized or the specific version is present: A single rule will fire at this point in both cases and the same actions will be executed.

2.6. Execution Model

The execution model can be described very briefly because it is based simply on running the type of production system simulation model already described. The simulation work used a current version of the "user-device interaction simulation" package shown in Figure 1 and described in Kieras and Polson (1985). In this simulation, the task specifications and the production rules are used by the production rule interpreter to carry out tasks, interacting with a simulated device. Based on the discussion in Kieras (1984), the execution model can be characterized as deterministic, specifying which processes and how many steps will be used for each task. The simulation provides predictor variables by counting the number of recognize-act cycles and the number and type of actions needed to perform each task. These actions may be complex operators defined for the particular task, simple operators such as pressing a key, or working memory operators that add or delete items in working memory. Performance on a task is measured in terms of the time to complete a task, and this time is assumed to be a function of the number of recognize-act cycles and the operators:

$$\textit{Execution time} = \textit{No. of cycles} \times \textit{time/cycle} + \textit{time spent in operators} \qquad (1)$$

This model is similar to the Card et al. (1983) Keystroke-Level Model with the mental operators replaced by the number of recognize-act cycles and the complex operators. One important difference from the principles described by Card et al. as characterizing the Model Human Processor is that the time for each recognize-act cycle of the performance model is assumed to be constant, whereas Card et al. proposed a cognitive processing rate that could be variable, affected by both task demands and the amount of practice. In the execution model, both experts and novices are assumed to have the same cycle time; differences in speed must be caused by differences in the number of cycles and the time to perform the complex operators. The assumption of constant cycle time, regardless of the contents of working memory or differences between subjects, has also been made in the simulation of reading processes developed by Thibadeau, Just, and Carpenter (1982; see also Just & Thibadeau, 1984).

3. EXPERIMENTAL EDITOR AND ITS COGNITIVE COMPLEXITY MODEL

In this section, the experimental editor is described and then its cognitive complexity model is presented as a complete example of a model constructed using the style rules. Although the editor used in the experiments was actually a simulated one, it was chosen to be realistic in its performance. This means that although it did not have the full range of functionality that a real editor would have, the functions that it did have are commonly used ones, such as inserting or deleting text, and the methods are those used by an actual editor.

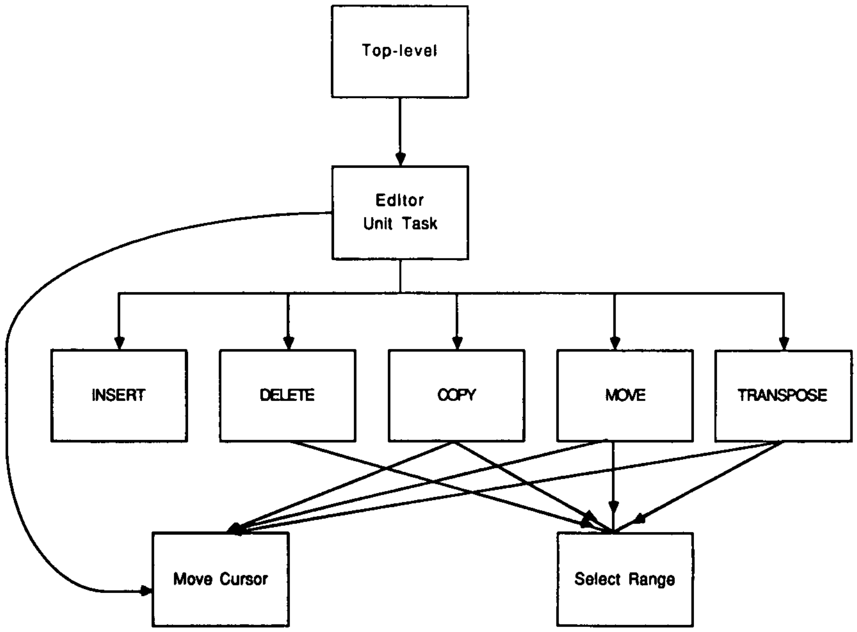
3.1. Editor Description

The editor used for these experiments was based on the IBM Displaywriter, greatly simplified by removing all menus and implementing only a subset of the possible editing functions: INSERT, DELETE, COPY, and MOVE. In addition, a new function was defined, TRANSPOSE, that, although it is not a realistic function, was intended to provide a relatively complicated function that still had definite transfer possibilities with the other functions. Cursor positioning in the experimental editor was done using cursor Arrow keys only; there was no find function.

To perform an editing function, the user first positions the cursor at the starting location of the edit and then indicates the operation to be performed by pressing the appropriate function key. The user then performs the appropriate actions for that function. At the end of the edit, the user can indicate that the edit was correct by pressing an ACCEPT key, or can undo the edit by pressing a REJECT key.

After pressing a function key, the following actions have to be carried out. For the INSERT function, the user simply types in the text to be inserted and, at the end, presses the ENTER key to indicate that insertion was complete. The DELETE, MOVE, COPY, and TRANSPOSE functions involve specifying the range of the operation by highlighting the corresponding text on the screen. This is done by moving the cursor across the text, either by using the cursor keys or by entering a single character, which causes the cursor to move to the next instance of that character in the text. An arbitrary sequence of character

Figure 9. **Method hierarchy for experimental editor.**



and cursor keys can be used to select the range. Range selection is terminated by pressing the ENTER key. During range selection, the editor prompts with a message such as "Delete what?"

For the DELETE function, after specifying the range, pressing the ENTER key causes deletion to take place. For both MOVE and COPY, once the range has been selected, the prompt message "To where?" appears, and the cursor keys are used to move to the target location, followed by pressing the ENTER key to perform the operation. For the TRANSPOSE function, the two target ranges to be transposed have to be specified. Once the first range is specified, the cursor keys are used to move to the beginning of the second range, then pressing the special T2 key indicates that the second range is to be selected. After selecting the second range, pressing the ENTER key causes the transposition of the two pieces of text.

3.2. Cognitive Complexity Model

In developing the cognitive complexity model for the experimental editor, the method hierarchy shown in Figure 9 was constructed, and then the specific steps for each method were generated. The production rule model was then written using both the hierarchy and the specific steps. This is a

Figure 10. **Steps of general methods in the experimental editor.**

Unit Task	Select Range	Move Cursor
Start unit task	Start select range	Start move cursor
Look up task location	Look up range end	REPEAT:
Do task function	Select end character	Figure distance to move
Finish unit task	Verify highlighting	Press arrow key
	Press ENTER	UNTIL: cursor is positioned
	Finish select range	Finish move cursor

conventional, structured programming approach to building the production rule representation. A preferable alternative would be to use the same information to first write a GOMS model in a notation similar to that of Card et al. (1983) or Kieras (1988). Many of the necessary decisions, such as how to decompose the user's knowledge into methods, would be made while writing the GOMS model. Then, the style rules could be applied to quickly produce the production rule representation.

Overview. The diagram in Figure 9 illustrates the hierarchy of methods. At the top is the top-level method that will acquire each unit task and add the goal of performing it until there are no more tasks to do. The editor UnitTask method moves the cursor to the start of the next unit task to be done and adds the goal of performing the method for that unit task. There are five possible specific editing methods that can be selected to do the edit in the unit task. In addition, there are the two general purpose methods; SelectRange, which is used by four of the five specific editing methods, and MoveCursor, which is used by the editor UnitTask method as well as three of the specific editing methods.

Figure 10 shows the individual steps of the methods for doing a unit task, for selecting the range, and for moving the cursor; Figure 11 shows the steps of the specific editing methods from Figure 9. The UnitTask method consists of locating the task, moving to the task location, and actually performing the task functions. Locating the task is done by looking in the manuscript for the location information, and the MoveCursor method is used to move to the task. Performing the task consists of first acquiring the task information by looking in the manuscript and then selecting and executing the editing function. If the method is DELETE, then it consists of pressing the DELETE key, checking the delete prompt, selecting the text to be deleted, checking the accept prompt, verifying that the task was performed correctly, and pressing the ACCEPT key.

Editor UnitTask Method. Figure 12 shows the representation of the unit task structure for text editing. Given the goal of performing the unit task, the

Figure 11. **Steps of specific editing methods in the experimental editor.**

Insert	Delete	Copy	Move	Transpose
Start insert	Start delete	Start copy	Start move	Start transpose
Press INSERT	Press DEL	Press COPY	Press MOVE	Press T1
Check prompt	Check prompt	Check prompt	Check prompt	Check prompt
Type in text	Select range	Select range	Select range	Select range
Verify insert	Verify delete	Check prompt	Check prompt	Look up start of second text
Press ACCEPT	Press ACCEPT	Look up where to copy to	Look up where move to	Move cursor
Finish insert	Finish delete	Move cursor	Move cursor	Press T2
		Press ENTER	Press ENTER	Check prompt
		Verify copy	Verify move	Select range
		Press ACCEPT	Press ACCEPT	Verify transpose
		Finish copy	Finish move	Press ACCEPT
				Finish transpose

first step, shown in UnitTask.P1, is to find the location of the next unit task and add the goal of moving the cursor there. The second step, UnitTask.P2, recognizes that the cursor is in the correct position so that the third step, UnitTask.P3, finds the information needed to select an editing method. When the specific method is complete, signaled by the appearance of the note, (NOTE TASK PERFORMED), the last rule fires, deletes the goal of performing the unit task, and returns control to the top level. If there are more unit tasks to do, then the top level adds the goal of performing a unit task and this method then runs again.

Example Representation of a Method. Figures 13 and 14 provide a complete example method for deletion, under novice user assumptions. Figure 13 shows the selection rule for this method. In the start rule, SelectDelete, the general goal to perform the unit task is (GOAL PERFORM UNIT-TASK). The STEP clause and the various NOTE clauses in the condition of this rule form the specific context. The rule action adds the goal of doing the DELETE method and a lockout note. The Finish rule, FinishDelete, will be triggered by the appearance in working memory of (NOTE DELETE DONE) as all the other condition clauses are already present. It deletes the notes about the specific context, the general goal to perform the task, and adds a note that the general goal has been satisfied.

In Figure 14, the rules NoviceDelete.P2 and NoviceDelete.P3 illustrate how the method for a subtask, namely, the SelectRange method is invoked. In NoviceDelete.P2, the goal of selecting the range is added to working memory. This assertion of the goal suffices to invoke the SelectRange method. When the method terminates, it adds a note to indicate the

Figure 12. **Editor unit task method.**

```
(StartUnitTask      IF  ((GOAL PERFORM UNIT-TASK)
                         (NOT (NOTE PERFORMING UNIT-TASK)))
                  THEN  ((Add STEP LOOKUP TASK-LOCATION)
                         (Add NOTE PERFORMING UNIT-TASK)))
(UnitTask.P1        IF  ((GOAL PERFORM UNIT-TASK)
                         (STEP LOOKUP TASK-LOCATION))
                  THEN  ((LookMSS TASK LOCATION)
                         (Add STEP MOVE CURSOR-TO-TASK)
                         (Add GOAL MOVE CURSOR)
                         (Delete STEP LOOKUP TASK-LOCATION)))
(UnitTask.P2        IF  ((GOAL PERFORM UNIT-TASK)
                         (STEP MOVE CURSOR-TO-TASK)
                         (NOTE CURSOR IN-POSITION))
                  THEN  ((Delete STEP MOVE CURSOR IN-POSITION)
                         (Add STEP LOOKUP TASK-FUNCTION))
(UnitTask.P3        IF  ((GOAL PERFORM UNIT-TASK)
                         (STEP LOOKUP TASK-FUNCTION)))
                  THEN  ((LookMSS FUNCTION ENTITY)
                         (Delete STEP LOOKUP TASK-FUNCTION)
                         (Add GOAL PERFORM EDIT-TASK)
                         (Add STEP DO TASK-FUNCTION)))
(UnitTask.Done      IF  ((GOAL PERFORM UNIT-TASK)
                         (STEP DO TASK-FUNCTION)
                         (NOTE EDIT-TASK PERFORMED))
                  THEN  ((Delete GOAL PERFORM UNIT-TASK)
                         (Delete STEP DO TASK-FUNCTION)
                         (Delete NOTE EXECUTING TASK)
                         (Delete NOTE EDIT-TASK PERFORMED)))
```

termination, and in NoviceDelete.P3 the presence of this note is tested for. In this way, the deletion method waits for the SelectRange method to finish before it continues its own processing.

The rule NoviceDelete.P2 in Figure 14 also illustrates how the user's perceiving a prompt on the screen is represented. The perceptual and reading comprehension processes involved in reading a prompt are not modeled; rather, when the device provides a prompt, the simulation adds a clause with the tag, DEVICE, directly into working memory where it can be tested by the condition of a production rule.

Figure 14 clearly illustrates the style rule that each overt action should be represented in a separate production rule. After the start rule, the first rule performs a keystroke, the second checks a prompt, the third waits for another method to be completed, the fourth checks another prompt, the fifth verifies that the appropriate text was deleted, and the sixth performs a keystroke. Each action has its own rule, and each rule contains a single user action.

Figure 13. **Example of novice method selection rules.**

```
(SelectDelete       IF   ((GOAL PERFORM EDIT-TASK)
                          (STEP DO TASK-FUNCTION)
                          (NOTE FUNCTION DELETE)
                          (NOTE ENTITY STRING)
                          (NOT (NOTE PERFORMING EDIT)))
                  THEN   ((Add GOAL DELETE STRING)
                          (Add NOTE PERFORMING EDIT)))
(FinishDelete       IF   ((GOAL PERFORM EDIT-TASK)
                          (STEP DO TASK-FUNCTION)
                          (NOTE PERFORMING EDIT)
                          (NOTE FUNCTION DELETE)
                          (NOTE ENTITY STRING)
                          (NOTE DELETE DONE))
                  THEN   ((Add NOTE EDIT-TASK PERFORMED)
                          (Add STEP FINISH TASK)
                          (Delete GOAL PERFORM EDIT-TASK)
                          (Delete STEP DO TASK-FUNCTION)
                          (Delete NOTE FUNCTION DELETE)
                          (Delete NOTE ENTITY STRING)
                          (Delete NOTE DELETE DONE)
                          (Delete NOTE PERFORMING EDIT-FUNCTION)))
```

Some methods, such as the MoveCursor method, were represented in the editor cognitive complexity model without selection rules. In the editor used in our experiments, such methods have no options, no context sensitivity, and are heavily used, and would thus seem to be stripped down to a minimum of rules. For example, although the subtask of moving the cursor appears in a variety of contexts, its execution is independent of where it was invoked. In addition, moving the cursor is a frequently performed method. All these characteristics of the MoveCursor method suggest that it does not need to be represented with selection rules. However, including such rules would not affect the pattern of the results (to be presented) and would produce a more consistent set of rules.

The results of applying the expert rule generation process to the method in Figures 13 and 14 are shown in Figure 15. The two prompt-checking steps were simply removed, and the remaining rules were collapsed together. Although all the rules through NoviceDelete.P2, including the selection rule, could be collapsed together into a single rule, SelectDelete, NoviceDelete.P3 contains the return point for the range-selection method and, therefore, cannot be collapsed with the preceding rules.

Transfer Between Methods. The transfer process was described in an earlier section (Section 2.5), and some example rules drawn from the

Figure 14. **Example of novice method.**

```
(StartNoviceDelete  IF ((GOAL DELETE STRING)
                        (NOT (NOTE EXECUTING DELETE)))
                  THEN ((Add STEP ACTIVATE-FUNCTION DELETE)
                        (Add NOTE EXECUTING DELETE)))
(NoviceDelete.P1    IF ((GOAL DELETE STRING)
                        (STEP ACTIVATE-FUNCTION DELETE))
                  THEN ((DoKeystroke DEL)
                        (Delete STEP ACTIVATE-FUNCTION DELETE)
                        (Add STEP CHECK-PROMPT DELETE)))
(NoviceDelete.P2    IF ((GOAL DELETE STRING)
                        (STEP CHECK-PROMPT DELETE)
                        (DEVICE USER MESSAGE DeleteWhat))
                  THEN ((Delete STEP CHECK-PROMPT DELETE)
                        (Add GOAL SELECT RANGE)
                        (Add STEP SELECT-RANGE DELETE)))
(NoviceDelete.P3    IF ((GOAL DELETE STRING)
                        (STEP SELECT-RANGE DELETE)
                        (NOTE RANGE SELECTED))
                  THEN ((Delete NOTE RANGE SELECTED)
                        (Delete STEP SELECT-RANGE DELETE)
                        (Add STEP CHECK-PROMPT ACCEPT)))
(NoviceDelete.P4    IF ((GOAL DELETE STRING)
                        (STEP CHECK-PROMPT ACCEPT)
                        (DEVICE USER MESSAGE PressAcceptOrReject))
                  THEN ((Delete STEP CHECK-PROMPT ACCEPT)
                        (Add STEP VERIFY DELETE)))
(NoviceDelete.P5    IF ((GOAL DELETE STRING)
                        (STEP VERIFY DELETE))
                  THEN ((Verify Task DELETE TEXT)
                        (Delete STEP VERIFY DELETE)
                        (Add STEP PRESS ACCEPT)))
(NoviceDelete.P6    IF ((GOAL DELETE STRING)
                        (STEP PRESS ACCEPT))
                  THEN ((DoKeystroke ACCEPT)
                        (Delete STEP PRESS ACCEPT)
                        (Add STEP FINISH UP)))
(NoviceDelete.Done  IF ((GOAL DELETE STRING)
                        (STEP FINISH UP))
                  THEN ((Delete GOAL DELETE STRING)
                        (Delete STEP FINISH UP)
                        (Delete NOTE EXECUTING DELETE)
                        (Add NOTE DELETE DONE)))
```

experimental editor were shown there in Figure 8. More details are illustrated by examining the transfer status of the novice deletion rules after the 33 rules for inserting a text have already been acquired.

First, note that two general methods, UnitTask and MoveCursor, have

Figure 15. **Example of expert method.**

```
(SelectDelete            IF    ((GOAL PERFORM EDIT-TASK)
                                (STEP DO TASK-FUNCTION)
                                (NOTE FUNCTION DELETE)
                                (NOTE ENTITY STRING)
                                (NOT (NOTE PERFORMING EDIT)))
                         THEN  ((Add GOAL DELETE STRING)
                                (Add NOTE PERFORMING EDIT)
                                (DoKeystroke DEL)
                                (Add GOAL SELECT RANGE)
                                (Add STEP SELECT-RANGE DELETE)))
(ExpertDelete.P1         IF    ((GOAL DELETE STRING)
                                (STEP SELECT-RANGE DELETE)
                                (NOTE RANGE SELECTED))
                         THEN  ((Delete NOTE RANGE SELECTED)
                                (Delete STEP SELECT-RANGE DELETE)
                                (Verify Task DELETE TEXT)
                                (DoKeystroke ACCEPT)
                                (Add STEP FINISH UP)))
(FinishDelete            IF    ((GOAL PERFORM EDIT-TASK)
                                (GOAL DELETE STRING)
                                (STEP DO TASK-FUNCTION)
                                (STEP FINISH UP)
                                (NOTE PERFORMING EDIT)
                                (NOTE FUNCTION DELETE)
                                (NOTE ENTITY STRING))
                         THEN  ((Add NOTE TASK PERFORMED)
                                (Add STEP FINISH TASK)
                                (Delete GOAL PERFORM EDIT-TASK)
                                (Delete GOAL DELETE STRING)
                                (Delete STEP DO TASK-FUNCTION)
                                (Delete STEP FINISH UP)
                                (Delete NOTE FUNCTION DELETE)
                                (Delete NOTE ENTITY STRING)
                                (Delete NOTE PERFORMING EDIT)))
```

already been acquired as part of the INSERT method. Thus, in processing the DELETE method, the transfer model designates all of the rules for these two methods as being identical to already learned rules, meaning that they transfer. However, the general range-selection method has not been already learned and, therefore, must be acquired during learning of the deletion method. So, all eight rules in the range-selection method are designated new. Figure 17, in the data analysis section that follows (Section 4.1), shows the total numbers of rules of each transfer status for each method for the different training orders used in the experiment. Note that the numbers for the same method are different, depending on what methods have already been learned.

Verifying and Running the Model. The complete cognitive complexity model involved a total of 99 production rules in the novice form and 71 in the expert form. The expert and novice rule sets were both tested by running them in the user–device interaction simulation package to ensure that they correctly performed all 112 editing tasks used in Experiment 2. Once the rule sets were fully tested and corrected, then the learning predictions were generated from the novice rule set, and the performance predictor variables were generated from both the novice and expert rule sets.

4. EVALUATION OF THE MODEL

To evaluate the model, two experiments were conducted to test its predictions. In the first, predictions from applying the transfer model were tested against subjects' performance as they learned the editor. In the second experiment, the execution model generated predictions for how long it should take to perform various editing tasks, both as a novice and then after practice. These predictions were tested against subjects' performance as they performed a set of editing tasks, after training and after 8 days of practice.

4.1. Experiment 1: Test of Learning Predictions

The purpose of this experiment was to test the predictions made by the model for the learning of the procedures. Three different training orders were chosen in order to maximize the differences predicted by the transfer model. Evaluating the model was done by using multiple-regression to fit it to the data, along the lines of Kieras (1984).

Method

Procedure. In the experiment, subjects first read instructions on the structure of the editing task and how to use the cursor positioning keys. They then learned the five text-editing methods. Subjects learned each method in two phases: First, subjects studied detailed instructions on how to perform the method. These instructions included information about how to select the range. Second, subjects practiced the method by editing a two-page manuscript with four edits on each page. The manuscripts were taken from the draft of an introductory psychology textbook and had been reverse edited so that when subjects completed a given edit the text made sense. If a subject made an error, appropriate feedback was given immediately after pressing the ACCEPT key, and the subject then reviewed the instructions for the method. Subjects were required to redo an edit if they made any errors. The learning criterion was one error-free repetition of all eight edits, whereupon subjects immediately went on to the instructions for the next method.

Figure 16. **Training order of procedures.**

	Serial Position				
	1	2	3	4	5
Order 1	INSERT	DELETE	COPY	MOVE	TRANSPOSE
Order 2	INSERT	TRANSPOSE	DELETE	COPY	MOVE
Order 3	INSERT	COPY	MOVE	TRANSPOSE	DELETE

Apparatus. The experimental environment consisted of two video terminals driven by programs running under DEC VAX VMS that communicated with each other. The first program implemented the simplified screen text editor, and the second was a computer-assisted instruction (CAI) package that presented all instructional material and provided feedback.

When the ACCEPT key was pressed, the editor sent a description of the edit just completed to the CAI package where it was evaluated. If the subject had made an error, the CAI package could determine what kind of error had been made, for example, typing in the incorrect text for an insert. The CAI package then gave feedback to the subject that indicated the type of error.

Subjects. Subjects were recruited from the Boulder, Colorado community by means of a newspaper advertisement; they were paid $15 for participating in the experiment. Subjects were required to be able to type and to have had no computing or word-processing experience. A total of 90 subjects participated in the experiment.

Design. Training-order condition was a between-subjects factor, with 30 subjects in each condition; each subject was randomly assigned to one of the three training-order conditions. Each subject learned all five editing functions in all three conditions. For each training order, the INSERT method was learned first, followed by the other four methods. The three training orders are shown in Figure 16.

Results and Discussion

Analysis Variables. The dependent variables were study time (the time spent studying the initial instructions for each editing method), practice time (the time from the end of study of the instructions until subjects reached criterion performance, including all error recovery and method review), and the total training time (the sum of the study and practice times).

For each method in each training order, the transfer model generated the total number of production rules (TOTALRULES), the number of new rules to be learned (NEW), the number of rules that were the same as an existing rule

Figure 17. **Number and type of rules to be learned in each training order.**

	Order 1				
	INSERT	*DELETE*	*COPY*	*MOVE*	*TRANSPOSE*
NEW	33	18	7	3	10
SAME	0	21	34	34	34
GENERALIZED	0	5	2	4	0
SUBSUMED	0	0	5	7	5

	Order 2				
	INSERT	TRANSPOSE	DELETE	COPY	MOVE
NEW	33	23	5	7	3
SAME	0	21	34	34	34
GENERALIZED	0	5	0	2	4
SUBSUMED	0	0	5	5	7

	Order 3				
	INSERT	COPY	MOVE	TRANSPOSE	DELETE
NEW	33	20	3	10	5
SAME	0	21	34	34	34
GENERALIZED	0	7	4	0	0
SUBSUMED	0	0	7	5	5

(SAME), the number of rules that could be generalized with an existing rule (GENERALIZED), and the number of rules subsumed under an existing generalized rule (SUBSUMED). These numbers are shown in Figure 17. Figure 17 shows that the number of new rules decreases rapidly, but not uniformly for the different training orders.

Analysis Results. The first analyses were performed using only the number of new rules as a predictor variable. This simple analysis parallels that in Polson and Kieras (1985). The regression equation for mean total training time as a function of the number of new rules is:

$$\textit{Total training time} = 683.2 \text{ s} + 34.7 \text{ s} \times (\textit{number of new rules}) \qquad (2)$$

This equation accounts for 88% of the variance among the 15 mean training times; the coefficient for the number of new production rules is about 30 s. This result, that the number of new production rules is a strong predictor of learning time, is comparable to the results obtained in Polson and Kieras (1985) and Kieras and Bovair (1986).

More extensive stepwise multiple-regression analyses were performed on

Figure 18. **Regression analyses of training time (N = 450).**

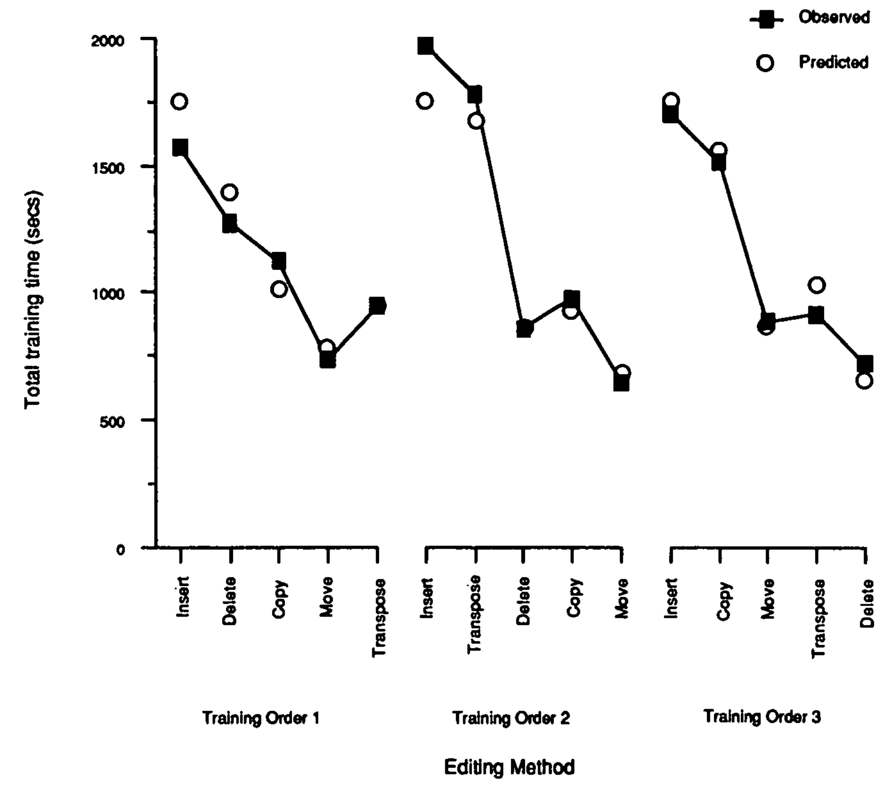
Variable	Final Coefficient	Final Standard Coefficient	F
	Total Training Time (R^2 = .50)		
CONSTANT	- 1200.35		
SMEAN	1.00	.40	140.56
NEW	33.77	.55	52.25
TOTALRULES	22.78	.20	13.09
ORDER	- 96.80	- .20	9.40
	Study Time (R^2 = .65)		
CONSTANT	- 181.40		
SMEAN	1.00	.46	277.01
NEW	5.95	.57	79.37
ORDER	- 21.36	- .26	122.44
TOTALRULES	3.63	.18	16.27
	Practice Time (R^2 = .44)		
CONSTANT	- 1018.94		
SMEAN	1.00	.40	128.63
NEW	27.82	.50	38.25
TOTALRULES	19.15	.18	9.98
ORDER	- 75.44	- .17	6.16

the individual subject data. The dependent variables were the three described previously (study time, practice time, and total training time) for each subject on each method. In addition to the predictor variables from the transfer simulation, the analysis included, as predictors, the subject's mean training time for all procedures (SMEAN) to handle the within-subject design (see Pedhazur, 1982) and the main effect of serial order (ORDER). For each dependent variable in the analysis, there were 450 data points, one for each subject on each editing function in each training-order condition.

The results of this analysis are shown in Figure 18, which shows the coefficients in the final equation for the variables that entered the stepwise analysis. The F ratios are the "F to remove" and provide the appropriate test of significance of each variable in the final equation under the assumption that each variable was the last to enter. In addition, the standardized regression coefficients allow comparisons of the importance of each variable to be made, independent of scale. The predicted and observed mean times for each training-order condition for the three dependent variables are shown in Figures 19, 20, and 21.

For total training time, about 50% of the variance in individual subject

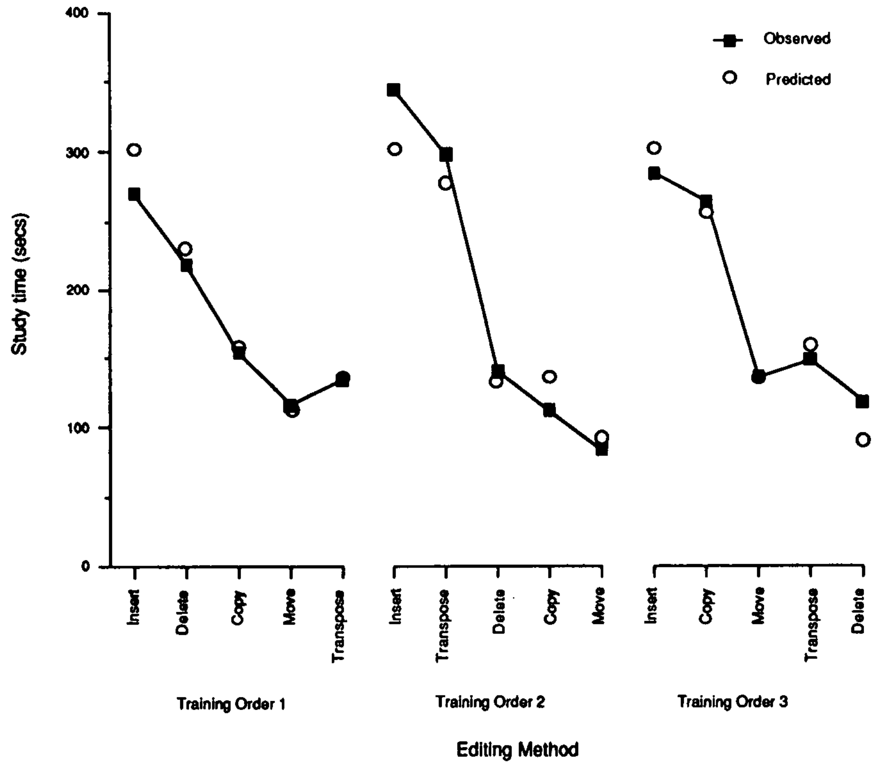
Figure 19. **Predicted and observed mean total training times for each training-order condition.**



time on each function is accounted for by the final equation. When the components of training time are analyzed separately, about 65% of the variance in individual subject study time and about 42% of the variance in practice time are accounted for by the final equations. The most important predictor variable for all three dependent variables is the number of new rules in each editing function (NEW). By itself, NEW accounts for 42% of the variance in study time, 26% of that of practice time, and 33% of total training time.

For total training time, the coefficient of TOTALRULES shows that each rule requires about 22.8 s, but each new rule requires an additional 33.8 s. Thus, there is a cost for each rule, because the subject had to read about and execute every step of the editing function, regardless of whether that step had already been learned or not. But each *new* rule to be learned requires a substantial additional amount of time, about 30 s. In addition to this effect of NEW, there was also a learning-to-learn effect as shown by ORDER. Each editing function

Figure 20. **Predicted and observed mean study times for each training-order condition.**

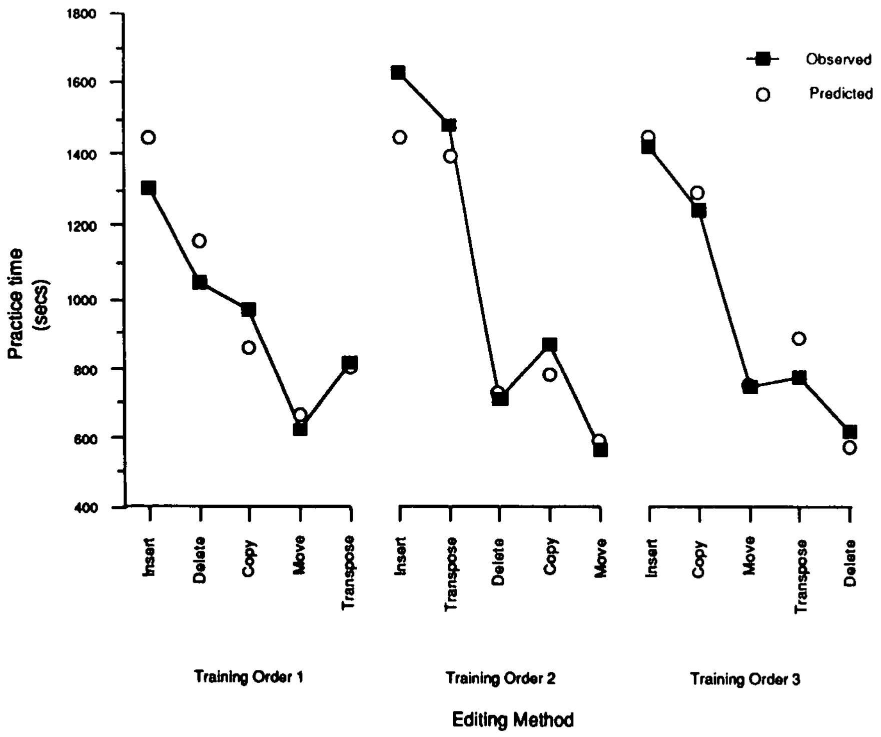


was learned about 97 s faster than the previous one, after the decreasing number of new rules was taken into account. The small standardized coefficient of ORDER, compared to NEW, makes clear that this is a relatively small effect.

The results for study time and practice time are comparable to that for total time; the standardized coefficients are similar in all three analyses. There is a suggestion that study time is more closely related to the predictors than practice time, although, as would be expected, most of the total training time can be attributed to practice, rather than study of the instructions.

A puzzling result, not apparent from Figure 18, is that NEW uniquely accounts for only 5% of the variance in practice time and 6% of both study time and total training time. This low proportion of variance uniquely accounted for is a result of the presence of ORDER in the equation. Figure 18 illustrates the relationship between ORDER and training time quite clearly; the later a method is learned, the faster it is learned. The curve in this figure is

Figure 21. **Predicted and observed mean practice times for each training-order condition.**



similar to a learning curve with a rapid monotonic decrease in time over trials. However, this is not the same task being repeated on each trial, as in a typical learning curve, but different tasks on each trial. Thus, the decrease in time for later methods does not simply reflect the effect of practice, but the effect of transfer in that a decreasing amount of new information must be learned for each new method. This means that there is a close relationship between training order and the amount of new information that must be aquired, as shown by the correlation of −.86 between ORDER and NEW. However, it is important to note that this is a characteristic of training order and the methods for this particular device. In fact, it could be argued that transfer functions such as this are characteristic of a consistent user interface.

It is quite possible to use training orders and devices where training order and amount of transfer are much less confounded, as in Kieras and Bovair (1986), in which NEW uniquely accounts for 47% of the variance even when ORDER is in the equation. In the present results, if ORDER is omitted then NEW uniquely accounts for 19% of the total training time (23% of study time, 15%

of practice). Also, NEW is a better predictor than ORDER. NEW has a larger standard regression coefficient (.55) than ORDER (−.20); in total training time, NEW accounts for 33% alone, 6% uniquely, whereas ORDER accounts for 28% alone, 1% uniquely. In addition, NEW is of more theoretical interest, both because it explains most of the ORDER effect, and because it can predict training times when the relation between training order and training time is nonmonotonic, as in Kieras and Bovair (1986) and in Polson (1987).

Summary. Despite the effects of learning to learn, the production system variables provided by the transfer model explain training times quite well. The number of new rules accounts for 33% of the variance in total training time by itself, and is a better predictor of total training time on a particular edit than is the subject's individual mean, which accounts for only 16% of the variance by itself. Thus, by analyzing these editing functions in terms of transfer of production rules, it is possible to account well for the relative difficulty of learning.

4.2. Experiment 2: Test of Performance Predictions

The second experiment was performed in order to test the performance predictions of the model. Subjects were trained to use the experimental editor as in Experiment 1, and then data were collected on their performance of editing tasks over the course of 8 days. The total time required by subjects to complete each edit was compared to the model predictions.

Method

Apparatus, Design, and Procedure. The experimental environment and instructional materials were the same as for Experiment 1. This experiment required 9 days to complete. On the first day, subjects were trained using the same training procedures as those in Experiment 1, with half using Training Order 1 and half using Training Order 3. On 8 practice days, subjects edited a new manuscript for each day, with each manuscript containing a total of 70 edits. Edits were randomized in blocks of 10 edits per page, with two of each of the five types on a page. Each day began with a brief review of the instructions for the five editing methods. The correctness of each edit was checked by the CAI package after the ACCEPT key was pressed. Incorrect edits were repeated.

Subjects. Eight subjects were recruited from the same community and in the same way as for Experiment 1. They were paid $15 for each day.

Not all of the subjects were used in the analysis. To properly compare the cognitive complexity model predictions to the task completion times, both the

Figure 22. **Predictor variables generated by the simulation.**

CYCLES	Number of recognize/act cycles required to complete the task
NLOOKMSS	Number of times LookMSS used
NVERIFYTASK	Number of times VerifyTask used
NFIGUREDISTANCE	Number of times FigureDistance used
NKEYSTROKES	Total number of keystrokes
ADD-<item>	Number of times an item was added to working memory
DELETE-<item>	Number of times an item was deleted from working memory
ORDER	Sequential order of edit (8–56)
EXPERT	Dummy variable for the effect of expertise
EXPNLOOKMSS	Interaction variables to test effect of expertise
EXPNVERIFY	
EXPORDER	

model and the subjects needed to use the same methods, defined as the same keystroke sequences. The problematic method was range selection. The criterion was that subjects should use the same range-selection method as the simulation, or a simple variant of it, on approximately 80% of the edits. The simulation selects the range by entering the last character of the range, repeatedly if necessary, until the correct range is highlighted. Seven of the eight subjects used this method; only one subject used the cursor keys exclusively to select the range of editing operations; this subject's data were dropped.

Results and Discussion

Analysis Variables. The dependent variable was performance time, defined as the time to perform a single editing task, extending from the last cursor keystroke prior to range selection until the ACCEPT key was pressed at the end of the edit. This time was averaged over subjects. The times for the INSERT method and cursor movement between editing tasks were not included because they were dominated by typing time. The first eight edits for each day's practice session were dropped. For simplicity, only the first and last days' data were included; on the first day subjects were assumed to perform as novices, and on the last day subjects were assumed to perform as experts.

Performing the same tasks, the simulation used the production rules to edit exactly the same text for both days as the subjects did. Various statistics were collected from these simulation runs in order to generate predictor variables. These predictor variables are shown in Figure 22.

An especially important predictor is the number of cycles required by the production system to execute each task. This was generated using both the novice and the expert rule sets. The variable, CYCLES, was defined such that

Figure 23. **Regression analysis on performance time (N = 48, R^2 = .79).**

Variable	Final Coefficient	Final Standard Coefficient	F
CONSTANT	1.597		
NLOOKMSS	3.395	.303	1.35
EXPERT	-.462	-.029	.01
NVERIFY	1.199	.289	11.59
ORDER	-.142	-.248	12.56
EXPNLOOKMSS	-1.952	-.389	2.61
EXPORDER	.186	.446	10.45
EXPNVERIFY	-1.297	-.409	7.95
CYCLES	.107	.531	59.48

for Day 1, CYCLES was the number of execution cycles for the novice rules, and, for Day 8, CYCLES was the number of execution cycles for the expert rules.

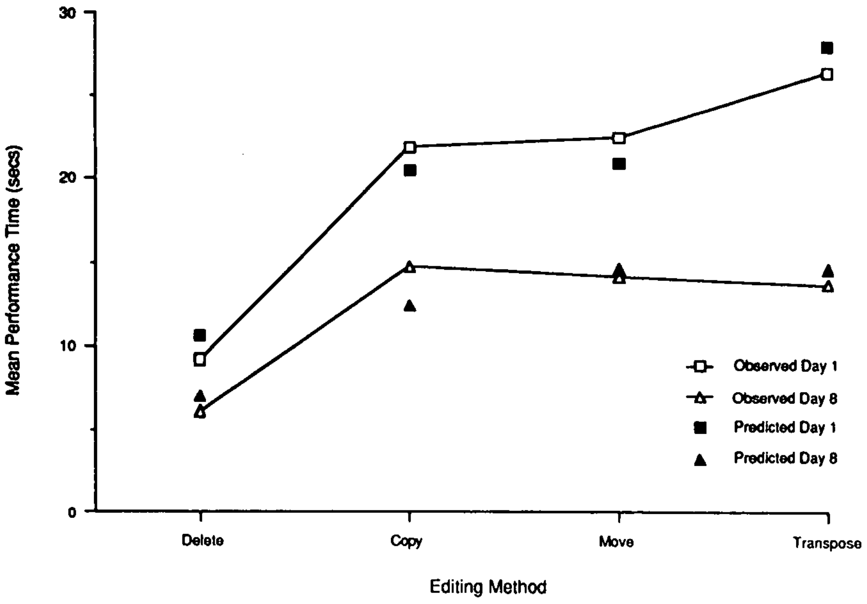
The simulation monitored the three complex operators, LookMSS, VerifyTask, and FigureDistance, to generate three corresponding predictor variables. These were NLOOKMSS, which is the number of times information had to be looked up from the manuscript; NVERIFY, which is the number of task verifications (e.g., verifying that highlighting was correct or that the correct material was deleted); and NFIGUREDISTANCE, which is the number of times that the distance of the current cursor location from the desired location was computed. Additional predictor variables were NKEYSTROKES, which is the total number of keystrokes used by the simulation to perform the task; and ORDER, which is simply the sequential order of the edit, with values ranging from 8 to 56.

In addition, predictor variables were generated to test for the effects of expertise over the course of the training period. The variable, EXPERT, was a dummy variable that was zero for Day 1 and one for Day 8. Several expertise interaction variables were defined (e.g., EXPNLOOKMSS, which was zero for Day 1 and had the value NLOOKMSS for Day 8). The simulation also generated variables that were used to test for effects of load on working memory; a discussion of the results for these variables follows.

Analysis Results. Figure 23 shows the multiple-regression analysis of performance time using the predictors generated by the simulation. This analysis includes both the times from Day 1 and the times from Day 8. Examination of Figure 23 shows that a total of 80% of the variance among the 96 means was accounted for. Predicted and observed means for each type of edit on Day 1 and Day 8 are shown in Figure 24.

There are several points of interest in the analysis shown in Figure 23. The

Figure 24. **Predicted and observed mean performance times for novices (Day 1) and experts (Day 8).**



first is that CYCLES has a coefficient of about one tenth of a second per cycle. This result is similar to the cognitive processor cycle time assumed in the Model Human Processor of Card et al. (1983). The preliminary analysis in Polson and Kieras (1985) yielded a comparable coefficient of .17, although there were some important differences in the role of memory load predictors (to be discussed).

The analysis also indicates some striking changes as subjects increased their expertise. The two actions of looking at the manuscript and verifying the task both take a substantial amount of time for novice users; the final coefficients are 3.4 s for NLOOKMSS and 1.2 s for NVERIFY. For the practiced user, the negative final coefficient of −1.952 for EXPNLOOKMSS indicates that the amount of time spent looking at the manuscript is reduced by about 2 s to only about 1.4 s, although this coefficient is not significant. The negative final coefficient for EXPNVERIFY shows that the time for task verifications is reduced by 1.3 s to about zero for practiced users. As suggested earlier in the style rules, this implies that practiced users do not check device prompts and do not constantly verify that their work is correct, whereas novices seem to do so.

The effect of practice within a session is shown by the variable, ORDER, whose value is simply the sequential order of the edit and is 8 for the first and 56 for the last edit of the day. As shown by the coefficient for ORDER, subjects

Figure 25. **Regression analysis using keystrokes on performance time ($N = 48$, $R^2 = .79$).**

Variable	Final Coefficient	Final Standard Coefficient	*F*
CONSTANT	.664		
NLOOKMSS	4.773	.426	26.52
EXPERT	.377	.024	.01
NVERIFY	1.117	.269	10.08
ORDER	−.133	−.232	11.10
EXPNLOOKMSS	−3.066	−.612	6.73
EXPORDER	.190	.454	10.81
EXPNVERIFY	−.371	−.433	8.81
NKEYSTROKES	.259	.480	59.81

speed up somewhat during their first session so that they perform each edit .14 s faster than the previous one. The variable, EXPORDER, shows that on Day 8 this slight practice effect had disappeared almost completely.

Comparison of CYCLES With Number of Keystrokes. To assess the value of CYCLES as a predictor variable, a similar analysis was performed using the number of keystrokes instead of the number of cycles; all other operator predictor variables were also used. Card et al. (1983) showed that the number of keystrokes was a good predictor of performance time, and CYCLES should be at least as good or even better. Note that NKEYSTROKES is the number of keystrokes needed to do the task as predicted by the simulation, not the number of keystrokes observed in the data. This model is essentially a form of the Card et al. (1983) Keystroke-Level Model.

The results of this regression analysis are shown in Figure 25. The final R^2 is .79, the same value as in the CYCLES analysis. The coefficient for NKEYSTROKES is .259, which is well in line with the figures reported by Card et al. (1983) of .20/keystroke for the average skilled typist and .28 for the average nonsecretary typist. An interaction variable was defined for expertise and number of keystrokes, but it did not enter the equation, meaning that there was no difference in keystroke times on Day 1 and Day 8.

The similar power of number of cycles and number of keystrokes as predictors is not surprising, given that the correlation between them is .921. When either one of the two is in the equation, the other does not contribute significantly. Some additional analyses showed that the number of keystrokes is not a particularly good predictor by itself, accounting for only 33% of the variance, compared to 58% for CYCLES by itself. It is only when given the benefit of an analysis that includes complex operators and expertise that the number of keystrokes becomes as good a predictor as CYCLES. Thus, despite the high correlation, CYCLES is a more useful and theoretically relevant predictor than the number of keystrokes.

Validity of Expert-Novice Distinctions. To test that the novice and expert rule sets were actually predicting performance differentially, an analysis similar to that shown in Figure 23 was performed, but with the CYCLES predictor reversed, so that the number of cycles derived from the novice rules was used for Day 8, and the number of cycles from expert rules was used on Day 1. In this case, CYCLES becomes a poorer predictor than keystrokes. This result, in conjunction with the intuitive appeal of the expertise results in the Figure 23 analysis, gives some support to the distinctions made between the expert and novice representations.

These results on modeling the expert-novice distinction suggest that a simple form of rule composition is a reasonable explanation of faster execution time with greater expertise. In addition, an important difference between practiced users and novices is that practiced users spend much less time looking at the screen to check for prompts or the accuracy of their work. This suggests that feedback to the user may only be useful to novices at an early stage of learning. On the other hand, the provision of feedback may not hurt users with greater expertise because practiced users learn not to pay attention to it.

Memory Load. A final point of interest, not immediately apparent from Figure 23, is that no memory-load variables, such as the number of items added to working memory, were significant predictors. These variables are highly correlated with other variables that appear in the Figure 23 analysis, such as CYCLES and NLOOKMSS. But it was decided to give CYCLES and counts of overt operators, such as LOOKMSS, priority in the prediction equation, and, as a result, the memory-load variables cannot account for additional variance in our study. The number of LOOKMSS operators, in particular, is highly correlated with the measures of memory load because performing a LOOKMSS to get the information needed to perform a task results in several NOTEs being added to working memory. Therefore, if the number of LOOKMSS operators is used as a predictor, the memory load variables will not contribute additionally to the regression equation.

The results described here found no effect of working memory load, but the Polson and Kieras (1985) preliminary model analysis found that the number of goals and notes added to working memory are an important predictor. One possible reason for this discrepancy is that the cognitive complexity model in Polson and Kieras did not follow the style rules described earlier, meaning that the structure of the two rule sets is somewhat different and, therefore, would be expected to generate different predictions. However, the difference between the rule sets is not very great. Rather, the probable reason why the Polson and Kieras analysis found an effect of memory load is that they did not use the number of operators such as LOOKMSS as predictor variables.

It is an interesting result that in an ordinary text editor, possible memory-load contributions to execution time are not apparent once cycles and complex operators are taken into account. In fact, the text-editing task may not be a good one for detecting such working memory-load effects. In this editor and these tasks, the maximum number of NOTE items in working memory at any one time is only six, well within the capacity of working memory, and so perhaps no effects should be expected.

5. GENERAL DISCUSSION

5.1. Conclusions

The results provide strong support for the cognitive complexity theory proposed by Kieras and Polson (1985). The predictions derived from the model predict both learning and execution data quite well, accounting for a substantial portion of the variance in both learning and execution time.

Another major piece of support for the theory comes from the work described in Kieras and Bovair (1986) and its essential agreement with the work described here. Although the two studies were made on different types of systems and used different paradigms, there is a striking similarity in the learning predictions. In both studies, the number of new rules to be learned is an excellent predictor of the learning time for a procedure—both the total learning time and the study time for the instructions. No learning overload effects due to the length of the first procedure were observed in this study, as they were in Kieras and Bovair, perhaps because a simple procedure (INSERT) was always trained first.

The correlation between cycles and keystrokes could be expected. In the task of text editing, the mental time required for a procedure is strongly dependent on the number of keystrokes that need to be made, and so this result is not surprising. As Card et al. (1983) suggested, to the extent that using a text editor is a routine cognitive skill (i.e., it does not involve much problem solving), there is not much cognitive activity preceding each keystroke, and thus the number of keystrokes should be a good predictor. However, describing methods just in terms of the number of keystrokes and complex operators along the lines of a Keystroke-Level Model does not make predictions about learning and transfer. In contrast, the production rule implementation of a GOMS model can be used to make predictions about both learning and performance.

5.2. Applying the Cognitive Complexity Approach

If cognitive complexity models can account for important aspects of learning and performance, they should be useful as a quantitative design tool, in the spirit advocated by Card et al. (1983). But as Kieras (1988) pointed out, there are problems with using the cognitive complexity approach as a design tool. One problem is the technical difficulty of constructing a production system model; because of the simplicity of its notation and the constraints imposed by the style rules, writing a PPS production rule model such as the one described in this article is probably as easy as writing a production system model could be. Note that many of the methods (e.g., cursor movement) could be shared between models for different systems, meaning that the cost of constructing new models would diminish as more were built. Even so, production rule modeling is potentially a difficult job for the nonspecialist. The solution that Kieras (1988) suggested was to develop a high-level GOMS modeling language that includes English-like statements that represent individual production rules. Thus, when constructed correctly, the high-level model has a specified relationship to the underlying production rule model, and the same quantitative predictions can be made. Thus, many of the benefits of generating a production rule model can be achieved more simply.

Thus, the purely technical problem of applying cognitive complexity theory has a straightforward solution. But by far the most difficult problem is conducting the GOMS-based task analysis that provides the information required for a cognitive complexity model. A proposed GOMS task analysis methodology is discussed at some length in Kieras (1988) but has yet to be tested and refined in practice.

Support. This research and preparation of this article was supported by the International Business Machines Corporation and the Office of Naval Research Cognitive Science Program. The opinions and conclusions stated in this article are those of the authors and not necessarily those of the sponsors.

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

Barnard, P. J. (1987). Cognitive resources and the learning of human-computer dialogs. In J. M. Carroll (Ed.), *Interfacing thought: Cognitive aspects of human-computer interaction* (pp. 112-158). Cambridge, MA: MIT Press.

Bovair, S., Kieras, D. E., & Polson, P. G. (1988). *The acquisition of text editing skill: A production system analysis* (Tech. Rep. No. 28). Ann Arbor: University of Michigan.

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

Covrigaru, A., & Kieras, D. E. (1987). *PPS: A parsimonious production system* (Tech. Rep. No. 26, TR-87/ONR-26). Ann Arbor: University of Michigan.

Green, T. R. G., & Payne, S. J. (1984). Organization and learnability in computer languages. *International Journal of Man–Machine Studies, 21,* 7–18.

Just, M. A., & Carpenter, P. A. (1987). *The psychology of reading and language comprehension.* Boston: Allyn & Bacon.

Just, M. A., & Thibadeau, R. H. (1984). Developing a computer model of reading times. In D. E. Kieras & M. Just (Eds.), *New methods in reading comprehension research* (pp. 349–364). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Kieras, D. E. (1982). A model of reader strategy for abstracting main ideas from simple technical prose. *Text, 2,* 47–82.

Kieras, D. E. (1984). A method for comparing a simulation to reading time data. In D. E. Kieras & M. Just (Eds.), *New methods in reading comprehension research* (pp. 299–325). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Kieras, D. E. (1988). Towards a practical GOMS model methodology for user interface design. In M. Helander (Ed.), *Handbook of human–computer interaction* (pp. 135–157). Amsterdam: Elsevier.

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. (1983). A generalized transition network representation of interactive systems. *Proceedings of the CHI '83 Conference on Human Factors in Computing,* 103–106. Boston, MA: ACM.

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.

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and performance VIII,* 693–718. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

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

Payne, S. J., & Green, T. R. G. (1986). Task-action grammars: A model of the mental representation of task languages. *Human–Computer Interaction, 2,* 93–134.

Pedhazur, E. J. (1982). *Multiple regression in behavioral research* (2nd ed.). New York: Holt, Rinehart & Winston.

Polson, P. G. (1987). A quantitative model of human–computer interaction. In J. M. Carroll (Ed.), *Interfacing thought: Cognitive aspects of human–computer interaction* (pp. 184–235). Cambridge, MA: MIT Press.

Polson, P. G., Bovair, S., & Kieras, D. E. (1987). Transfer between text editors. *Proceedings of the CHI '87 Conference on Human Factors in Computing,* 27–32. Toronto: ACM.

Polson, P. G., & Kieras, D. E. (1985). A quantitative model of the learning and performance of text editing knowledge. *Proceedings of the CHI '85 Conference on Human Factors in Computing,* 207-212. San Francisco: ACM.

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,* 78-83. Boston: ACM.

Simon, H. A. (1969). *The sciences of the artificial.* Cambridge, MA: MIT Press.

Singley, M. K., & Anderson, J. R. (1987-1988). A keystroke analysis of learning and transfer in text editing. *Human-Computer Interaction, 3,* 223-274.

Thibadeau, R. H., Just, M. A., & Carpenter, P. A. (1982). A model of the time course and content of reading. *Cognitive Science, 6,* 157-203.

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

HCI Editorial Record. First manuscript received December 28, 1987. Revisions received June 27, 1989 and October 3, 1989. Accepted by Thomas Moran. —*Editor*