Int. J. Man-Machine Studies (1985), **22**, 365-394

An approach to the formal analysis of user complexity†

DAVID KIERAS
Electrical Engineering & Computer Science Department University of Michigan, Ann Arbor, MI 48109, USA

AND

PETER G. POLSON
Psychology Department, University of Colorado, Boulder, CO 80309, USA

(*Received 5 February 1983, and in revised form 3 November 1983*)

A formal approach to analysing the *user complexity* of interactive systems or devices is described, based on theoretical results from cognitive psychology. The user's knowledge of how to use a system to accomplish the various tasks is represented in a procedural notation that permits quantification of the amount and complexity of the knowledge required and the cognitive processing load involved in using a system. Making a system more usable can be accomplished by altering its design until the knowledge is adequately simplified. By representing the device behaviour formally as well, it is possible to simulate the user-device interaction to obtain rigorous measures of user complexity.

1 Introduction

1.1. OVERVIEW

The purpose of this paper is to present a detailed and precise framework for future work on *user complexity*, which is the complexity of a device or system from the point of view of the user.[1] This paper will focus on the development of a formal description of the user's knowledge of how to use the device, and of the behaviour of the device itself. It will also suggest how these formal descriptions can be used, especially in the form of computer simulation models, to define the user complexity of a device precisely.

Section 2 presents a formal description system for the user's knowledge of how to use the device; this can be used to quantify certain aspects of user complexity. Several examples of how the system can represent *how-to-use-it* knowledge are provided, based on the IBM Displaywriter. In addition, the nature of the user's knowledge of *how the device works* will be discussed briefly, and a proposal made for how the desirable qualities of this knowledge can be defined using the description of the knowledge of how to use the device.

To describe user complexity fully and usefully, it is essential to be able to describe the device in a way that can be related to the knowledge required to operate it. Section 3 presents a formal description system for representing interactive devices, followed by a discussion of how the relationship between the device and the how-to-use-it

† This paper is a product of a collaborative project conducted by the authors at the University of Michigan and the University of Colorado, with support from the International Business Machines Corporation. The opinions and conclusions contained within this research are those of the authors and not necessarily those of IBM.

[1] Since this paper went to press, we have decided to use the term *cognitive complexity* instead of *user complexity*.

0020-7373/85/040365+30$03.00

knowledge can be described by the two representation systems. The final section of the paper will present the conclusion that these representations of the device and user will be of great practical value when combined with computer simulation techniques.

1.2. REVIEW OF PREVIOUS WORK

As proposed in Kieras & Polson (1982), the complexity of a device, from the point of view of a user, depends on the amount, content, and structure of the knowledge required to operate the device successfully. Furthermore, for a new user, complexity is also determined by the difficulty of acquiring the new knowledge necessary for this purpose. Two major components of the knowledge involved in operating a device are the *user's task representation* and the *user's device representation.* The task representation is the user's knowledge of how to carry out a task using the device. The device representation consists of the knowledge that a user has about the device itself.

1.2.1. The user's task representation

The user's task representation is assumed to be described by a model developed by Card, Moran, & Newell (1980, 1983), the GOMS model. This representation is composed of a user's understanding of *goals*, *operations*, *methods*, and *selection rules.*

Goals define a decomposition of the task into more elementary component tasks that can be accomplished using the device. Goals and subgoals are both hierarchically and sequentially related to each other, so that the complete *goal structure* specifies the temporal and conceptual relationships between the various component tasks. Thus, the goal structure can be considered to be a plan for carrying out the complete task.

Operations are mental representations of the various elementary functions that the device can perform, and of other cognitive operations that occur during the execution of the task.

Methods are procedures for satisfying specific goals and subgoals. In the simplest form, a method would consist of a single keystroke. However, methods can be complex, in that they can consist of a hierarchy of goals and subgoals, along with the operations necessary to satisfy each component of the goal structure defined by a specific method.

Selection rules specify what method should be used to satisfy a given goal or subgoal in a specified context. When there are several methods with different characteristics that can be used to satisfy the same goal, selection rules pick out the method that is most appropriate to the specific context.

In Kieras & Polson (1982), the user's *job environment* was defined as a collection of *task environments.* An "environment" in this sense consists of the externally-imposed requirements that a person must satisfy, and the equipment available for use in meeting these requirements [see Kieras & Polson (1982) for further discussion]. Here it is assumed that the user's knowledge of the job environment is the same kind of information as is contained in the representation of a task environment. Therefore, rather than further distinguishing between the job situation and the tasks that appear in the job situation, this paper considers them jointly, as the *job–task environment.*

The user's *job–task representation* is the user's knowledge of the job–task environment. This includes the user's understanding of the relationships between the job and the various task environments, and of the task environments themselves. The job-related portions of the representation will be represented by only two components of the GOMS model. The *job goals* consist simply of accomplishing successfully the various tasks comprising the job environment. The *job selection rules* enable users to correctly choose the appropriate task knowledge to fulfill a job goal. As in Kieras & Polson

(1982), each of the task representations consists of all four components of the GOMS model, which are representations of the goals, operations, methods, and selection rules involved in the task.

A distinction can be made between the *device-dependent* and the *device-independent* knowledge in the user's task representation (Kieras & Polson, 1982). This distinction can be illustrated using the task of entering a heavily marked-up manuscript into a word processor like the IBM Displaywriter. Much of the knowledge required to perform this task is device-independent, in that the same knowledge would be necessary to carry out the task on any word processor or even on a typewriter. Examples include knowledge about the format of the particular manuscript, the ability to interpret the notation used to indicate modifications in the rough draft, and so on. Device-dependent knowledge consists of knowledge of functions that are part of the task only because a particular device is being used. For example, different word processors require different procedures for deleting material from a document.

This distinction is important in the design of a new device. If the new user can apply device-independent knowledge, the new system will be easy to learn. If the design of the new device entails a large amount of device-dependent knowledge, such as the first word processors to use floppy disks, which have complicated maintenance procedures, then the device will be relatively hard to learn.

1.2.2. The user's device representation

Kieras & Polson (1982) assume that there are four categories of information in the user's knowledge of the device itself: (1) *Task-relevant knowledge* is information about the goals that the device can be used to satisfy, the operations that can be performed on the device, and the operating procedures for the device. Such knowledge of the device is the counterpart to the user's task representation. (2) *Device layout knowledge* is knowledge concerning the physical layout of the device, such as the location of controls, the format of the display, and the location of various switches and status indicators. (3) *Device behaviour knowledge* is the user's understanding of the relationships between the operation of various controls and the external behaviour of the device. (4) *How-it-works knowledge* is the user's understanding of the internal structures and functions of the device. Such knowledge is assumed to enable a user to generate explanations about the operating procedures and the behaviour of the device, and about the device-dependent components of the task representation.

1.2.3. Sources of user complexity

Kieras & Polson (1982) concluded that the complexity of a device is determined by the complexity of the knowledge representations required to operate that device. More specifically, the complexity of a device depends upon:

(1) The complexity of the user's task representation, and the learning, memory, and processing capacity demands implied by the task representation;

(2) The number of device-dependent functions, which are not part of the user's initial task representation, and the difficulty of learning them;

(3) The ease with which a user can acquire how-it-works knowledge.

1.3. APPROACH

The goal of this paper is to develop a rigorous quantitative analysis of user complexity. Achieving this goal requires suitable descriptions of both the user and the device. The remainder of this paper is concerned with defining two representations, one of the user's knowledge of how to accomplish the task, and the other of the device itself.

These representations will be defined so that the user, as represented in the user's task representation, can interact with the device as represented in the device representation. Furthermore, the interaction of the user and the device can be simulated on a computer using these representations, thus making it possible to investigate the user complexity issues involved in the interaction, without having an actual device available. Scientifically, this allows the development of a formal theoretical specification of the factors that contribute to user complexity. It also allows a rigorous test of whether a hypothesized representation of the task is correct. If it is, the simulated interaction should be successful. Once an accurate representation of the task is available, it can be examined to determine quantitatively what aspects of the device, and the task requirements, are producing undesirable user complexity. These quantitative measures of user complexity should correspond to actual human performance in using different devices. If so, these measures could be used to *predict* the usability of a proposed device, once the empirical relations between the measures and actual performance have been assessed. This empirical work to validate the approach and estimate parameter values is still in progress, and, consequently, is not described in this paper.

In what follows, the user's task representation will be defined in some detail; this is mainly a matter of defining a notation system that can be used to describe how-to-do-it knowledge of the various tasks that a person might try to accomplish using any device. The examples in this paper are based on the analysis of a word processor, the IBM Displaywriter. A representation system for the device will also be presented, which allows one to describe the behaviour of any interactive device. The logic of the two notation systems is that the outputs of the user's task representation constitute the inputs to the device representation, and the outputs of the device representation are inputs to the user's task representation. The following sections of the paper will present the features of both of these representation systems in some detail. Finally, we shall return to a discussion of the value of these representations and simulation models.

2. The user's job–task representation

2.1. PRODUCTION SYSTEMS

This section presents a formal notation system to represent the user's knowledge of the job-task environment, based on the *production system* concept. The initial segment of this section presents a brief introduction to production systems. The next segment describes the particular production system notation that is used here to represent the users' job-task knowledge, followed by some sample descriptions of the knowledge involved in operating the Displaywriter.

2.1.1. Background of the production system concept

Production systems were proposed by Newell & Simon (1972) as tools for building formal models of psychological processes. This formalism has been used to develop theories of problem solving processes (Newell & Simon, 1972; Simon, 1981; Karat, in press), text comprehension (Kieras, 1982*a*), learning (Anderson, 1982), and other cognitive processes. An excellent summary of the history, advantages and technical properties of production systems can be found in Anderson (1976).

The production system concept can be viewed as a modern form of a very old concept in psychology: behaviour, or mental processes, can be represented as a set of

specific response actions, each made in response to a particular stimulus condition. The traditional form of this concept in experimental psychology is Stimulus-Response Theory. The current theory of the learning of skills based on production systems (Anderson, 1982), is simply a very general, powerful, and detailed form of this traditional view.

The use of production systems also permits a very elegant theoretical formulation for the difference between the knowledge of facts, *declarative knowledge*, and the knowledge of how to do things, *procedural knowledge*. In Anderson's (1976) formulation, declarative knowledge is represented as a set of propositions organized as a semantic network, while procedural knowledge consists of production systems. Thus, the use of production systems to represent the user's knowledge of how to use a device is ideally suited to tie work on human-computer interaction issues into the mainstream of cognitive theory.

2.1.2. Basic principles of production systems

A production system is composed of a collection of *production rules* and a *working memory*. The working memory contains representations of the system's current goals, other information about the status of current and past actions, and representations of inputs from the environment.

A production rule, or *production* for short, is a condition-action pair of the form:

IF (*condition*) THEN (*action*)

The *condition* of a production is a statement about the contents of working memory, such as the presence or absence of specified goals, or specified inputs from the environment. If the condition is true, then the production is said to *fire*, and the *action* component of the production is executed. The action can consist of one or more elementary actions. Actions include the insertion and deletion of information such as goals from working memory or operations on the environment.

A set of production rules is essentially a program; the process of executing the program also needs to be specified. The production systems used in this paper operate by alternating between *recognize* and *act* modes of operation. During the recognize mode, the conditions of all productions are tested against the contents of working memory. The system then goes into the act mode, during which the action components of all of the productions that fired are executed. The system then returns to the recognize mode, the contents of its working memory having been modified; this leads to the opportunity for different productions to fire and so on. Programming in terms of production rules amounts to manipulating the contents of working memory to cause the pattern of production firings that will produce the desired sequence of actions.

The formalism presented here makes no attempt to represent the more fundamental cognitive processes, such as those that comprehend the manuscript in a word-processing task. Furthermore, the system at present does not have any knowledge of the semantics of its various goals, notes, or other operations that appear in elementary conditions or actions. Current cognitive theory provides some answers for these issues, but dealing with them at this time would greatly increase the complexity of the formalism, without contributing to the understanding of user complexity.

The relationship between the GOMS model and the production system formalism is direct. Goals are directly represented in a production system for job-task knowledge.

They appear in the conditions of almost all production rules, and are manipulated in many production actions. Methods appear in the form of sequences of production rules whose first member is triggered by the assertion of the goal of the method. The structure of subgoals incorporated in a method appears in the form of actions in the method productions that assert and delete goals. Selection rules are simply the production rules that control the execution of the methods. Thus, a selection rule can be a single production rule that asserts the goal that triggers the execution of a method, or it can be a collection of productions evaluating the current context in order to select the best method for that context. Operations are also scattered through the whole system, and consist of the different elementary actions, as well as environment-testing conditions.

2.1.3. Description of notation[2]

The notation presented here includes terms that are important for the analysis of word-processing tasks; the operation of other types of devices may require other terms. The examples in Tables 1–7 should be examined while reading the following description of the notation. The notation has many syntactic features of LISP, the language used for the simulation. Each production rule is a list of five terms, enclosed in parentheses. In order, the terms are a name for the production rule, the word IF, the condition, the word THEN, and the action. The condition and action are also lists and so are enclosed in parentheses.

Normally, the condition is made up of several elementary conditions combined with a LISP AND function, which means that all of the elementary conditions must be satisfied before the production can fire. An elementary condition is a test for the presence of a specified piece of information in working memory or in the environment. Each elementary condition consists of a function name that specifies the type of information involved in the test, followed by the specific pattern of information being tested for.

TEST-GOAL and TEST-NOTE conditions test for patterns in working memory. TEST-MSS tests for patterns in the manuscript. If there is a matching pattern, then the elementary condition is true. TEST-CURSOR, and some other functions, interrogate the state of the word processor. For example, (TEST-CURSOR 5 2) is true if the cursor is in column 5 of row 2 on the screen. Notice that some elementary conditions are negated with a NOT function.

Variables, indicated by a prefix of %, can appear in a pattern defining an elementary condition. When the condition is evaluated, if a variable is bound to a string, that string substitutes for the variable in the pattern. If the variable has a null value, it will match any part of a pattern, and will then be assigned a new value during the match process. For example, if there was a pattern "A B C" in working memory, and "A %X C" was an elementary condition pattern, and %X equalled null, then the patterns would match and %X would be bound to "B". The pattern ??? is a "wild card" which will match any term.

The action portion of a production rule is a list of elementary actions, which add and delete goals and notes, operate keys and controls on the device, and search for information on the screen or in the manuscript.

The ADD-GOAL, ADD-NOTE, DELETE-GOAL, and DELETE-NOTE actions add or delete the specified patterns, either goals or notes, from working memory. The

[2] Since this paper went to press, the notation used in our simulations has been simplified.

GET-NEXT-UNIT-TASK and LOOK-MSS actions scan for the specified information in the manuscript. These operations can also assign values to variables to save the result of a scan. The DO-KEYSTROKE and TYPE-IN actions specify actual operations on the device. The WAIT action means that after completion of the current production system cycle, the production system will wait for the device to respond to the keystrokes. Finally, PRINT-MSG is used to signal the operator of the simulation, and STOP-NOW is used to terminate the simulation run.

2.1.4. Examples of job-task knowledge

In this section, the production system formalism described in the preceding section is illustrated with an analysis of some of the components of a possible user's job-task representation for the IBM Displaywriter. Note that the remainder of this paper has many observations on the IBM Displaywriter and its documentation. It should be kept in mind that these remarks are based on a theoretical analysis, the correctness of which is not yet known and must be demonstrated in the laboratory.

The first example of job-task knowledge, shown in Table 1, illustrates job environment selection rules and goals. A secretary has been given the task of typing a clean first draft of a manuscript of a long journal article. Both a typewriter and Displaywriter are available. Two goals are in working memory, that of typing the new document and selecting the equipment.

TABLE 1

Examples of selection rules in the job representation

```
(Journal-article
IF (and (TEST-MSS manuscript is new journal article)
        (TEST-GOAL type manuscript)
        (TEST-GOAL select equipment))
THEN ((ADD-NOTE many revisions will be done)) )

(Use-Displaywriter
IF (and (TEST-GOAL type manuscript)
        (TEST-GOAL select equipment)
        (TEST-NOTE many revisions will be done)
        (TEST-MSS manuscript is long))
THEN ((ADD-GOAL use displaywriter)
        (ADD-GOAL type new manuscript into displaywriter)
        (DELETE-GOAL select equipment)) )
```

The two productions in Table 1 result in the secretary choosing to use the Displaywriter. They are a small sample of the many production rules that make up the user's knowledge of what task environments should be entered in a job situation.

The next series of examples comes from analysis of the processes involved in editing a marked-up draft of a manuscript already in the Displaywriter. The analysis very closely parallels that of Card *et al.* (1980, 1983), who assume that the task of editing a manuscript is broken up into a sequence of *unit tasks.* 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 for that unit task. The examples shown

in Tables 2–7 are excerpts from an actual working simulation of a user performing deletion functions on an IBM Displaywriter.

The two productions shown in Table 2 set up the next unit task in the manuscript. The action GET-NEXT-UNIT-TASK, in the first production, represents the process of scanning the manuscript for the next unit task; thereafter, TEST-MSS and LOOK-MSS functions will refer to a description of this unit task. If the end of the manuscript is reached, the editing session is terminated.

TABLE 2
Top level of editing task representation

```
(SET UP-UNIT-TASK
IF (AND (TEST-GOAL edit manuscript)
        (NOT (TEST-GOAL perform unit task)) )
THEN ((GET-NEXT-UNIT-TASK)
      (ADD-GOAL perform unit task)) )

(FINISHED-WITH-EDIT
IF (AND (TEST-GOAL edit manuscript)
        (TEST-NOTE no more tasks))
THEN ((DELETE-GOAL edit manuscript)
      (DELETE-NOTE no more tasks)
      (PRINT-MSG "I AM FINISHED EDITING THE MANUSCRIPT")
      (STOP-NOW)) )
```

Table 3 illustrates the selection rules and the accompanying control rules for three different deletion methods. Each selection rule, such as SELECT-CHARACTER-DELETION, is triggered by the appropriate pattern in the unit task description in the manuscript, and adds the goal of accomplishing the corresponding method. When this goal is accomplished and then removed by the rules for the method, a control production, such as CHARACTER-DELETION-DONE, is triggered to acknowledge that the method has been done. This action then leads to a firing of the first Table 2 rule, to set up the next unit task.

The next examples describe, at the level of the actual keystrokes, the methods for deleting characters, words, and arbitrary strings from a manuscript. These examples are based directly on the description of the DELETE command contained in the *Displaywriter System Operator Reference Guide*, Textpack 1, (IBM, 1981, pp 3-36 to 3-40). The examples do not include the cursor positioning selection rules and methods.

The production rules in Table 4 delete a single character at a specified location. The first rule is triggered if the cursor is not at the position specified in the unit task description, and adds the goal of moving the cursor to that position. The rules for cursor movement are not shown. When the cursor is properly located, the second rule fires and performs the keystrokes that delete the character.

A very similar set of productions, shown in Table 5, delete a word terminated by a space. This set of productions, based directly on the manual, will not work properly if the word to be deleted is terminated by a nonblank character.

The productions shown in Table 6 comprise a general method for deleting an arbitrary string from the manuscript. The first character of the string is located at column

TABLE 3
Selection and control rules for deletion methods

```
(SELECT-CHARACTER-DELETION
IF (AND (TEST-GOAL perform unit task)
        (TEST-MSS function is delete)
        (TEST-MSS entity is character)
        (NOT (TEST-GOAL delete character))
        (NOT (TEST-NOTE executing character deletion)) )
THEN ((ADD-GOAL delete character)
      (ADD-NOTE executing character deletion)
      (LOOK-MSS task is at %UT-HP %UT-VP)) )

(CHARACTER-DELETION-DONE
IF (AND (TEST-GOAL perform unit task)
        (TEST-NOTE executing character deletion)
        (NOT (TEST-GOAL delete character)) )
THEN ((DELETE-NOTE executing character deletion)
      (DELETE-GOAL perform unit task)) )

(SELECT-WORD-DELETION
IF (AND (TEST-GOAL perform unit task)
        (TEST-MSS function is delete)
        (TEST-MSS entity is word)
        (NOT (TEST-GOAL delete word))
        (NOT (TEST-NOTE executing word deletion)) )
THEN ((ADD-GOAL delete word)
      (ADD-NOTE executing word deletion)
      (LOOK-MSS task is at %UT-HP %UT-VP)) )

(WORD-DELETION-DONE
IF (AND (TEST-GOAL perform unit task)
        (TEST-NOTE executing word deletion)
        (NOT (TEST-GOAL delete word)) )
THEN ((DELETE-NOTE executing word deletion)
      (DELETE-GOAL perform unit task)) )

(SELECT-STRING-DELETION
IF (AND (TEST-GOAL perform unit task)
        (TEST-MSS function is delete)
        (TEST-MSS entity is string)
        (NOT (TEST-GOAL delete string))
        (NOT (TEST-NOTE executing string deletion)) )
THEN((ADD-GOAL delete string)
     (LOOK-MSS task is at %UT-HP %UT-VP)
     (ADD-NOTE executing string deletion)) )

(STRING-DELETION-DONE
IF (AND (TEST-GOAL perform unit task)
        (TEST-NOTE executing string deletion)
        (NOT (TEST-GOAL delete string)) )
THEN ((DELETE-NOTE executing string deletion)
      (DELETE-GOAL perform unit task)) )
```

TABLE 4
Method for deleting a single character

```
(PDELC1
IF (AND (TEST-GOAL delete character)
        (NOT (TEST-GOAL move cursor to %UT-HP %UT-VP))
        (NOT (TEST-CURSOR %UT-HP %UT-VP)) )
THEN ((ADD-GOAL move cursor to %UT-HP %UT-VP)) )

(PDELC2
IF (AND (TEST-GOAL delete character)
        (TEST-CURSOR %UT-HP %UT-VP))
THEN ((DO-KEYSTROKE DEL)
      (DO-KEYSTROKE ENTER)
      (WAIT)
      (DELETE-GOAL delete character)
      (UNBIND %UT-HP %UT-VP)) )
```

TABLE 5
Method to delete a single word

```
(PDELW1
IF (AND (TEST-GOAL delete word)
        (NOT (TEST-GOAL move cursor to %UT-HP %UT-VP))
        NOT (TEST-CURSOR %UT-HP %UT-VP)) )
THEN ((ADD-GOAL move cursor to %UT-HP %UT-VP)) )

(PDELW2
IF (AND (TEST-GOAL delete word)
        (TEST-CURSOR %UT-HP %UT-VP))
THEN ((DO-KEYSTROKE DEL)
      (DO-KEYSTROKE SPACE)
      (DO-KEYSTROKE ENTER)
      (WAIT)
      (DELETE-GOAL delete word)
      (UNBIND %UT-HP %UT-VP)) )
```

%UT-HP and row %UT-VP, and the last character of the string is located at column %S-HP and row %S-VP. Again, cursor-positioning selection rules and methods are not shown. The second rule adds the goal of selecting the range for the deletion. This is done by the rules in Table 7, which define a general range selection method and are called by other methods.

The deletion method shown in Tables 6 and 7 is an example of a *productive representation* of a method as described by Kieras & Polson (1982) in their discussion of the Displaywriter DELETE function. However, note that this general method is far more complicated than the specific methods shown in Tables 4 and 5. Execution of the general method involves far more productions than the specific ones, and also requires that the general cursor positioning selection rules and methods be used twice: first, in order to locate the beginning of the string, and second, to locate the end of the string.

TABLE 6
Method to delete an arbitrary string

```
(PDELSTR1
IF (AND (TEST-GOAL delete string)
        (NOT (TEST-CURSOR %UT-HP %UT-VP))
        (NOT (TEST-NOTE doing part 2 of string deletion))
        (NOT (TEST-GOAL move cursor to %UT-HP %UT-VP)) )
THEN ((ADD-GOAL move cursor to %UT-HP %UT-VP)) )

(PDELSTR2
IF (AND (TEST-GOAL delete string)
        (TEST-CURSOR %UT-HP %UT-VP)
        (NOT (TEST-GOAL select range ??? ???)) )
THEN((DO-KEYSTROKE DEL)
      (WAIT)
      (ADD-NOTE doing part 2 of string deletion)
      (LOOK-MSS string end is at %S-HP %S-VP)
      (ADD-GOAL select range %S-HP %S-VP)) )

(PDELSTR3
IF (AND (TEST-GOAL delete string)
        (TEST-NOTE range selected))
THEN (DO-KEYSTROKE ENTER)
      (WAIT)
      (DELETE-GOAL delete string)
      (DELETE-NOTE range selected)
      (DELETE-NOTE doing part 2 of string deletion)
      (UNBIND %UT-HP %UT-VP %S-HP %S-VP)) )
```

TABLE 7
Method to select a range

```
(PSELRAN1
IF (AND (TEST-GOAL select range %SRHP %SRVP)
        (NOT (TEST-CURSOR %SRHP %SRVP))
        (NOT (TEST-MSS end character is ???))
        (NOT (TEST-GOAL move cursor to %SRHP %SRVP)) )
THEN ((ADD-GOAL move cursor to %SRHP %SRVP)) )

(PSELRAN2
IF (AND (TEST-GOAL select range %SRHP %SRVP)
        (NOT (TEST-CURSOR %SRHP %SRVP))
        (TEST-MSS end character is %ENDCHAR))
THEN (TYPE-IN %ENDCHAR)
      (WAIT)
      (UNBIND %ENDCHAR)) )

(PSELRAN3
IF (AND (TEST-GOAL select range %SRVP)
        (TEST-CURSOR %SRVP)
THEN ((DELETE-GOAL select range %SRHP %SRVP)
      (ADD-NOTE range selected)
      (UNBIND %SRVP)) )
```

2.2. MEASURES OF COMPLEXITY IN THE JOB-TASK REPRESENTATION

This section presents some examples of design and analysis issues that the production system notation helps to clarify, and suggests some metrics based on the notation. These provide a definite and quantitative characterization of the user complexity of a device.

The DELETE function on the Displaywriter is documented in the IBM training materials in two basically different ways. One is as the generalized delete function, whose production rule representation is shown above in Tables 6 and 7. The other consists of a set of specialized procedures, such as those for deleting single characters or single words. These are represented in Tables 4 and 5. One could doubt the wisdom of teaching either one of these two approaches to the new user. That is, the general method corresponds closest to how the device actually operates, and knowledge of it will allow the user to delete any string efficiently. In contrast, the specialized delete functions are "buggy"; the delete-word procedure only works if the word is terminated by an undesired blank. Furthermore, the IBM training material presents several such specialized procedures. Why should the user have to learn several specific procedures instead of one general one?

Possible clarification appears when one considers the goal structures entailed by the two alternatives. Figure 1 shows the goal structures for the general delete function and

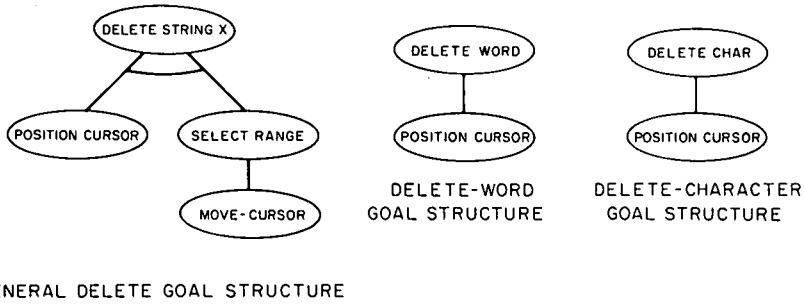

FIG. 1. Goal structures for three different deletion procedures.

the specialized delete-character and delete-word functions. These goal structures were constructed by examining the production rules presented above, and noting where goals are asserted, deleted, or tested; this information defines the hierarchy of goals and subgoals. These structures are represented using the standard AND-OR graph notation developed in artificial intelligence to represent problem-solving search spaces (Nilson, 1971). The node at the top represents the highest-level goal; to fulfill this goal, the lower level goals must all be met. Branches with an arc across them represent an AND structure; both subgoals must be met before the higher goal can be fulfilled. The standard way to process such structures is a depth-first left-to-right order. Hence, the general delete function requires that to fulfill the goal of deleting a specified string, one must first fulfill the goal of moving the cursor to the begining of the string, and then fulfill the goal of specifying the range of the function. Doing this requires first moving the cursor to the end of the string.

Comparing the two sets of goal structures shows some interesting tradeoffs between the general delete and the specialized delete procedures. The general delete has a goal structure that is three levels deep, while the specialized functions are only two levels deep. Since the list of current goals has to be kept in working memory, it is clear that the production rules for the general delete will impose a higher load on working memory than the specialized ones. Thus, once the procedures are learned, it appears that the specialized rules will impose less processing load than the general one. Furthermore, the delete-character and delete-word functions will probably account for almost all of the deletions that the typical user will make. With experience, therefore, these specialized procedures will become highly automated (Anderson, 1982). The documentation presenting these procedures may be providing the valuable service of getting the user started early on the very procedures that will be most valuable and most useful to have automated. Thus, the production system notation provides a useful quantification of working memory load in terms of the amount of goal information that must be processed.

Another quantification of the procedures consists of counting the production rules involved. For the procedures in Tables 4, 5, 6, and 7, the general delete procedure requires a total of six production rules, while each of the specialized procedures requires only two rules. Thus, the specific rules involve less learning of rules, and are also quicker to execute, since fewer rules are involved. Furthermore, they should automate quickly compared to the general rule. One might, however, have to learn a larger set of such rules. It could be predicted that a user provided with only the general rule would not achieve a high level of performance as quickly as one provided with the specialized rules. More detail on this example appears below.

2.3. THE RELATION BETWEEN THE JOB-TASK REPRESENTATION AND THE USER'S MODEL OF THE DEVICE

2.3.1. User's models and mental models

Kieras & Polson (1982) argued that having how-it-works knowledge of a device could have important effects on a user's ability to learn and operate the device. A *user's device model* is the person's understanding of the internal structure and functions of a device, which is how-it-works knowledge. *Mental models*, a more popular but vague, term for the user's device model, are a current topic of research (Gentner & Stevens, 1983). There is much confusion at present, concerning the characteristics of mental models; this is relieved somewhat by Norman's (1982) distinction between the *target system* (the actual device), the *conceptual model* of the target system (a formal model of the device), and the user's mental model.

There is considerable controversy concerning the content and structure of mental models. One view is that mental models are powerful representations that permit individuals to mentally simulate the possible actions of a system or device and thus predict its behaviour. The other end of the continuum, adopted here, is that mental models can consist of an analogical, incomplete, and sometimes very fragmentary, understanding of the how-it-works knowledge for a device (cf. Norman, 1982).

There is a reasonable degree of unanimity in the literature, concerning the good features of having mental models, although until recently little empirical data on the subject has been collected. Mental models permit individuals to acquire meaningful explanations of various device functions. Thus, the knowledge contained in a mental

model of a device is a source of meaningful explanations of various components of an individual's procedural knowledge contained in the job-task representation. Mental models also enable users to generate productive representations of various procedures. Kieras & Polson (1982) presented an analysis of the DELETE function in the Displaywriter which argued that, in order to understand the most general form of the delete operation, the user had to have some understanding of how the manuscript was represented internally.

Thus, mental models for devices, or user device models, are assumed to facilitate the novice's initial attempts to operate the device and to learn its operating procedures. A new user's initial attempts to work with a device are under the control of problem-solving processes (cf. Anderson, 1982). Having a mental model of the device leads a new user to consider only those sequences of operations that are consistent with the device model. A well-constructed device model will therefore facilitate a user's initial interactions with the device, by enabling the user to anticipate the proper sequences of actions correctly. Later in learning, having a device model will allow the user to reconstruct, by inference, sequences or steps that have been forgotten, leading to better performance during the middle portions of learning. It might seem that late performance, after considerable practice, would be unaffected by having a device model; however, the ability to deal with novel situations should be better if the user has a device model. Just such a pattern of results was observed by Kieras & Bovair (1983) in an experiment in which subjects learned (or inferred) the procedures for operating a simple control panel device, either with or without having learned a description of how the device worked.

Apparently, the computer industry feels that how-it-works knowledge should not be included in the documentation for devices for non-expert users. For example, the documentation for the IBM Displaywriter and proposals within IBM for improving the documentation of devices and programs explicitly *exclude* how-it-works knowledge and other information that will facilitate formation of coherent device models (Betke, Dean, Kaiser, Ort, & Pessin, 1981). We speculate that, in the past, documentation that presented extensive how-it-works knowledge caused difficulties for new users because they do not have the technical background necessary to understand detailed information about the structure and function of a program or device. On the other hand, recent work suggests that individuals construct mental models for all aspects of their environment, including devices (Kieras, 1982*b*). Omitting the information necessary to construct a powerful and effective device model will make it likely that the average user will form an incomplete and incorrect device model.

2.3.2. Criteria for selecting device model information

If having a device model that explains how the device works is beneficial to the user, then conveying such a model to the user during training should result in faster learning and better performance. However, the how-it-works information to be conveyed to the user must be selected so that is is actually relevant to the user's task. Kieras & Bovair (1983) argue that the relevant knowledge is that which enables the user to infer the *exact* operating procedures for the device. The problem is selecting this information out of the great quantity of possible information.

The how-it-works information for a device can be viewed as a hierarchy of explanations, in which each level is an elaboration providing more detail on the concepts

appearing in the level above. Such structures are well understood in the cognitive psychology of comprehension and memory. The question is, which portions of the hierarchy of how-it-works explanations should be selected for presentation to the new user. The speculation offered here is that this can be decided by using properties of the user's task representation. Basically, the criteria determining the explanations that should be provided depend on the relationship between explanations in the explanation hierarchy and goals in the user's task representation. Three preliminary criteria are as follows:

(1) If a goal was already part of the new user's previous task representation, then it need not be explained. For example, the goal of revising a document from a marked-up draft need not be explained to a new user of a word processor.

(2) If a goal appears in the new user's task representation that is unique or specific to the device, and did not appear in the user's previous task representation, then that goal must be explained by a corresponding level in the explanation hierarchy. For example, the goal of making a back-up copy of a diskette should be explained to a new user of word-processing systems in terms of how-it-works knowledge of magnetic storage systems.

(3) If an explanation does not correspond to a user goal, it should not be presented. For example, most computer hardware and software is arranged so that when a disk write operation is performed, the system automatically reads the information back and checks it for accuracy. This technical detail should not be presented to the user because there is no user goal that corresponds to this process, which is beyond the control of the user. That is, "WRITE ON DISKETTE AND CHECK FOR ACCURACY BY READING IT BACK" is not a user goal. This rule therefore specifies the maximum level of detail required for a task-relevant explanation of how a device works.

3. The device representation

In order to characterize the interaction between the user and the device, an explicit and formal representation of the *behaviour* of the device itself is needed. Just as the user can be represented with the job-task representation production system, the behaviour of the device can be represented with a formal notation as well. The relationships between the user's procedural knowledge and the behaviour of the device can then be formally represented. This representation of the device should have the following properties: (1) the representation should have well-defined formal properties, so that its correctness can be explicitly determined; (2) the representation should make *interactive* systems easy to represent; (3) the representation should be modular, so that as much or as little of the device can be represented as desired; (4) hierarchical control or structural relations should be easily represented; (5) the representation should not be committed to any particular hardware or software implementation; (6) it should be easy to represent features of the system that have psychological implications. This section of the paper introduces a formalism for describing the behaviour of a device.

3.1. PREVIOUS WORK ON FORMAL DEVICE REPRESENTATIONS

3.1.1. Grammatical representations

Previous attempts to introduce formal description systems for interactive systems have taken two forms. The first is based on the formalism known as Backus-Naur Form

(BNF), which consists of grammatical rewrite rules. It was introduced as a method of describing interactive systems by Reisner (1981, 1982) and others (Foley & Wallace, 1974; Embley 1978; Foley, 1980). The BNF form has the advantage that it is very familiar to computer users, because it is the conventional method used to describe the syntactical rules for programming languages. However, like the grammatical rewrite rules that appear in formal linguistics, it is most useful for expressing the syntax of a system, and not its semantics. In the context of an interactive system, the semantics of the system are related to the purposes and goals of the user, and to the behaviour of the system itself. The syntax, in contrast, only reflects the specific rules for the form of the commands. Consequently, as pointed out by Reisner (1981), the BNF representation itself permits interactions that are *non sequiturs*, or are meaningless, to the system. However, Reisner could use the BNF description to predict which of two functionally identical systems would be the easier one to use.

A further problem is that the structure of the system is not very clear when represented in BNF form. More specifically, the fact that the system may have several submodes or substates, and that its hardware and software may be layered in a hierarchical fashion, is not very apparent. Nevertheless, the hierarchical structure of a system is closely related to the goals and purposes that the user has in mind. This is because the designer of the system also had these goals and purposes at least partially in mind, and therefore structured the system to reflect them. Thus, it is important to have a way of clearly indicating the hierarchical structure of the system.

3.1.2. Transition network representations

The other formal description system that has often been used is the transition network. Transition network representations for systems are also familiar to many computer users, since state transition diagrams often appear in manuals, describing the various states or modes that an operating system can be in, and how transitions can be made between these states.

The transition network is actually a graphic representation of a *finite state machine*, which is a very general formal way of describing a system. A finite state machine, or finite *automaton*, consists of three finite sets and two functions. There is a set of states, a set of inputs, and a set of outputs. There is a state transition function which maps combinations of the previous state and the current input to a new state. There is an output function which maps combinations of previous states and inputs to outputs, thereby mapping transitions between states to an output [see Arbib (1969) and Minsky (1967) for more detail].

The basic idea of describing an interactive system in terms of a finite state machine is that the inputs to the system come from the user, and the outputs of the system go to the user. The states of the machine correspond to the modes or states of the interactive system. This application commonly appears in user documentation for interactive systems and devices, and has been presented as a general representation approach by workers such as Parnas (1969).

The transition network representation of a finite state machine is a diagram that consists of a series of *nodes*, and a series of labelled *arcs* that interconnect these nodes. The nodes represent states that the system can be in, and the arcs represent ways in which the system can make a transition from one state to the other. Each arc is labelled with the input that causes the transition, and also with the output that will be produced when the transition is made.

The problem with the transition network representation is that, at least in the published forms, it is not powerful or comprehensive enough to capture the important aspects of an interactive system. There are two limitations. First, the simple finite state machine simply can not represent in a comprehensible way the extremely large number of states that an interactive system can actually be in from the point of view of the user. For example, if one is using an editor, then every possible configuration of contents in the edit buffer is actually a distinct state of the system. Thus, if an editor is represented completely as a single finite state system, the number of possible states is astronomically large, so that the representation would be too cumbersome to be useful. Second, the simple transition network can not represent the hierarchical structure of a system, or any processes that appear repeatedly in the system. For example, most time-sharing systems have a single interface process for interacting with the terminal, which is involved whenever input from the user is required. A simple transition network would have to represent this subroutinized function by duplicating the same pattern of nodes and arcs throughout the network. Again, this makes the representation too clumsy to be useful.

3.1.3. Augmented recursive transition networks

These problems can be solved by the use of *augmented recursive transition networks.* Basically, an augmented transition network system is a finite state machine with registers. A register can be defined as a subsystem that stores a symbol string which can be manipulated or tested by the rest of the system. For example, the edit buffer in an editor can be considered as a register. Thus, the input set for an augmented machine consists of either inputs from the user, or some test on the contents of one or more registers. The output set for the machine consists both of user outputs that are signals to the user, or some manipulation of the symbols in one or more of the registers. The use of registers makes it possible to segment off certain state sets as being register states, thus simplifying the rest of the representation.

A *recursive* transition network is one in which one network can "call" another network as a subroutine; thus, networks can be embedded, or *nested,* within one another. By nesting networks, a very complicated system can be described as a hierarchy of simple subsystems, further simplifying the representation.

The original application of augmented recursive transition networks was by Woods (1970), as an extremely powerful way of stating parsing rules for natural language processing. Jacob (1982) suggested using Woods-style networks to describe an interactive system in a fairly clear fashion. However, such networks are most suitable for language processing because the nesting rules used in them are most naturally suited for recognizing patterns in input. On the other hand, the concept of *mode* of the system is not captured very naturally.

3.2. GENERALIZED TRANSITION NETWORKS

This paper introduces a more general form of transition network, named the generalized transition network (GTN). Like an ordinary transition network diagram, a GTN consists of nodes, which represent states, interconnected by arcs, which represent possible transitions between states. In diagram form, the nodes are represented by circles, and the arcs are represented by arrows between the circles (see top panel of Fig. 2). An arc consists of a condition, an action, and a specified next state. The arcs appear in a specified order; in the diagrams, this order is clockwise around the circle that represents

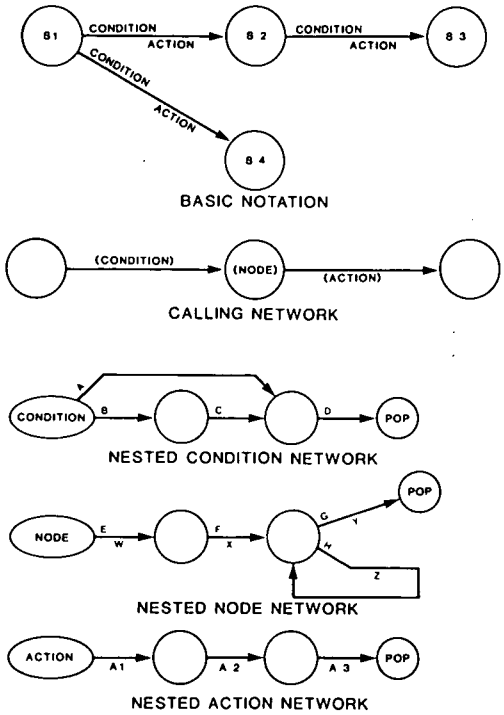

FIG. 2. Examples of notation for generalized transition network diagrams.

the state, beginning at the top. The condition is written above the arc, the action is written below the arc. The next state is simply the circle to which the arrowhead points.

3.2.1. GTN interpretation

The rules for interpreting a GTN can be stated rigorously in the form of a recursive function whose flowchart is shown in Fig. 3. However, these rules can also be presented informally as follows:

If the system is in a certain state, the arcs leaving that state are examined one at a time, in the specified order. If the condition on the arc is true, the system makes the transition specified by the arc. This consists of performing the action associated with the arc, and then entering the next state. The system then tries to leave the next state in the same manner.

Conditions and actions are optional. If there is no condition specified on an arc (a *null condition*), then the arc is crossed unconditionally when it is tested. Such arcs must always be the last arc associated with a state. If there is a *null action*, no action is taken when the arc is crossed.

If the system can not leave a state because there is no arc whose condition is satisfied, then it goes back over the previously followed arc, and tries a new arc from the previous state. If the system can not find a successful transition somewhere, then it is said to *fail*. If it does find a path through to a final terminal state, it *succeeds*.

The conditions and actions can consist of arbitrary tests or manipulations of an arbitrary number of registers of arbitrary capacity. Such registers are used to do the

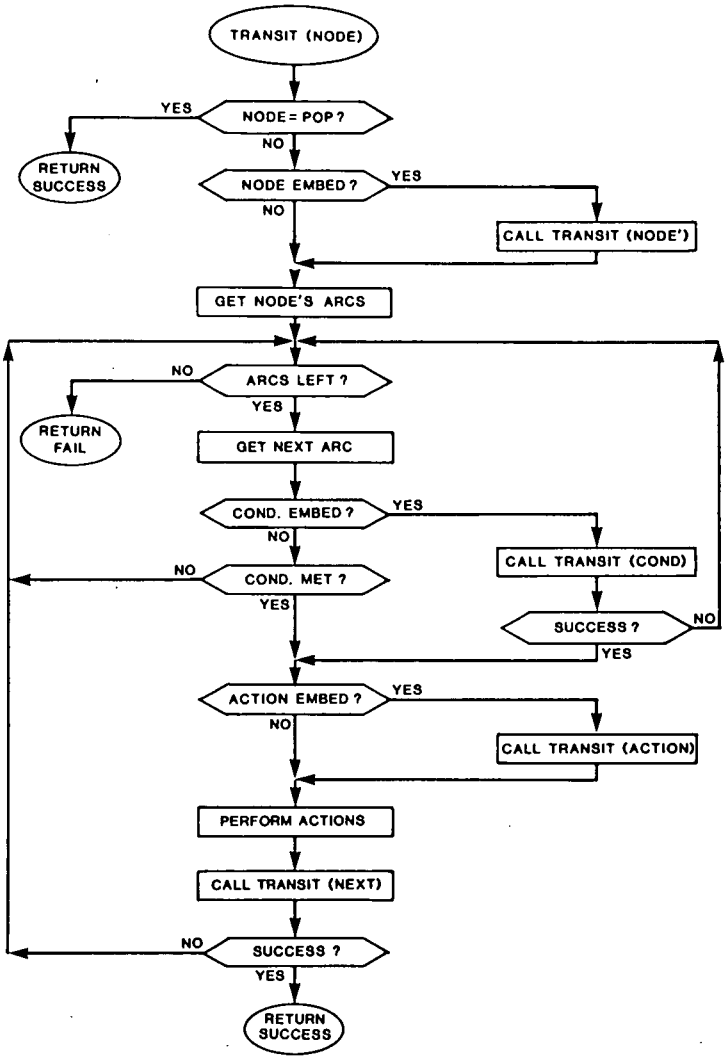

FIG. 3. Definition of how a GTN is interpreted in terms of a recursive function.

actual "work" of the system. Technically, this property gives the GTN Turing computation power, meaning that it can represent any computational system.

3.2.2. Nesting in GTNs

Nesting allows one GTN to call another in a manner analogous to subroutine calls in ordinary programming languages. This allows the representation to reflect the natural hierarchical structure of the system, and allows similar functions to be represented only once, and then called as a subroutine wherever needed. In a GTN, nesting can appear in three places: in *conditions*, in *actions*, and in *states*. These are illustrated in the bottom portion of Fig. 2, and are further described below. In all three cases, if a network is not allowed to call itself, or can call itself only a finite number of times, a

nested system is computationally equivalent to a system without nesting. If so, then for all three kinds of nesting, there is a simple rule, which need not be discussed here, for removing the nesting by constructing an equivalent system that does not have any nesting. Thus, all three forms of nesting are equivalent in computational power.

Condition nesting. In condition nesting, the name of a subnetwork appears in parentheses as the condition on an arc. In order to see if the condition is satisfied, the subnetwork is performed. If the subnetwork succeeds, then the condition is satisfied and the transition is made. The subnetwork succeeds if it makes a transition to a special terminal state called POP. This is analogous to a subroutine return statement in an ordinary programming language.

Action nesting. In action nesting, the name of a subnetwork appears in parentheses as one of the actions on an arc. When the action is executed the named subnetwork is performed, followed by any other primitive actions associated with the arc. The action subnetwork must always enter a POP state, so that the calling network can complete the transition.

State nesting. State nesting is indicated by the name of a subnetwork appearing in parentheses within a state node. Whenever the state is entered, the subnetwork is performed immediately, before any arcs on the calling state are tested. The subnetwork must always finish with a POP. The arcs leaving the calling state are then tested and followed.

3.2.3. General properties of GTNS for interactive systems

Unlike language-processing ATNs, GTNs that represent actual interactive systems are characterized by always having a successful, but not necessarily convenient, set of transitions available. For example, incorrect input sends the system into some error state, or one can always cancel out to the top level, or one can always give up and turn the machine off.

Apparently, GTNs for interactive systems rarely involve true recursion, in which a network calls itself directly or indirectly. This property is critical for the ATN work on parsing, but is apparently not important for interactive systems.

Nesting within actions seems to be mainly useful for describing computations, performed by the system, that do not involve user interaction. Thus, it does not usually need to be used if the concern is only with the user-device interaction. Nesting within conditions is mainly useful for specifying what pattern of input might be expected at a certain point. Nesting within states seems to be very valuable for representing modes of a system. For example, one can say that at the top level, the Displaywriter is in "REVISE" mode, which would be represented as a single state in the top-level GTN. However, this single state actually consists of a whole subsystem, represented by a hierarchy of GTNs embedded in that state. This form of nesting corresponds to the traditional use of state diagrams in computer documentation to describe the different modes of a system.

3.2.4. An example GTN

To illustrate how GTNs can be used to describe an interactive system, this section presents a GTN which is a simplified form of portions of the Displaywriter.

The top-level GTN shown in Fig. 4 that begins in a state labeled START is associated with the main task menu. It makes use of a subnetwork, INPUT, which handles input

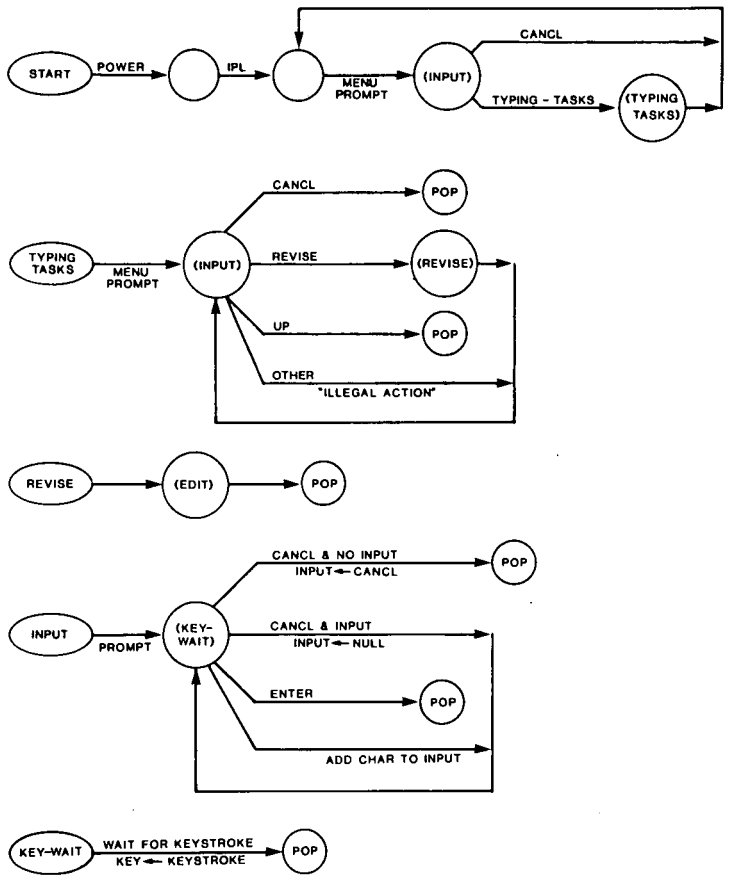

FIG. 4. Simplified GTN for the Displaywriter. The network should be read beginning with the node labeled "START" at the top.

from a prompt. The top-level GTN enters a state for TYPING TASKS if the user chooses that menu response.

The GTN for TYPING TASKS is associated with a second menu, and a state for REVISE can be entered. The embedded GTN for REVISE is drastically simplified here; it merely enters a GTN called EDIT. The EDIT GTN shown in Fig. 5 represents the top level of the editor. Each function is shown here in the form of embedded condition GTNs, although embedded states may have been more convenient.

Figure 6 shows the simple GTN for the top level of the delete function, and the more complex move function. The MOVE GTN first calls a SELECT-TARGET GTN, which is shared by the DELETE GTN, and which is used to select the material to be moved (or deleted), and then calls a MOVE-CURSOR GTN that represents the various ways the cursor can be moved with the MOVE command. Note that the FIND GTN can be called here as well as the top level of EDIT. The MOVE-CURSOR GTN is also shared with SELECT-TARGET.

The main point of this example is that a complex interactive system can be represented in a way that is not only easy to comprehend, but that also represents the structure of

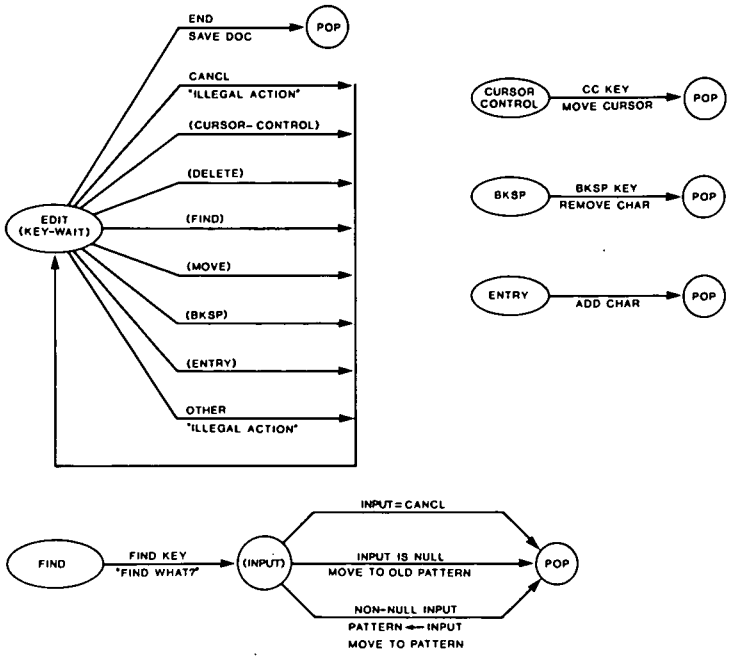

FIG. 5. Simplified GTN for the Displaywriter. The top level of the editor is the node labelled "EDIT".

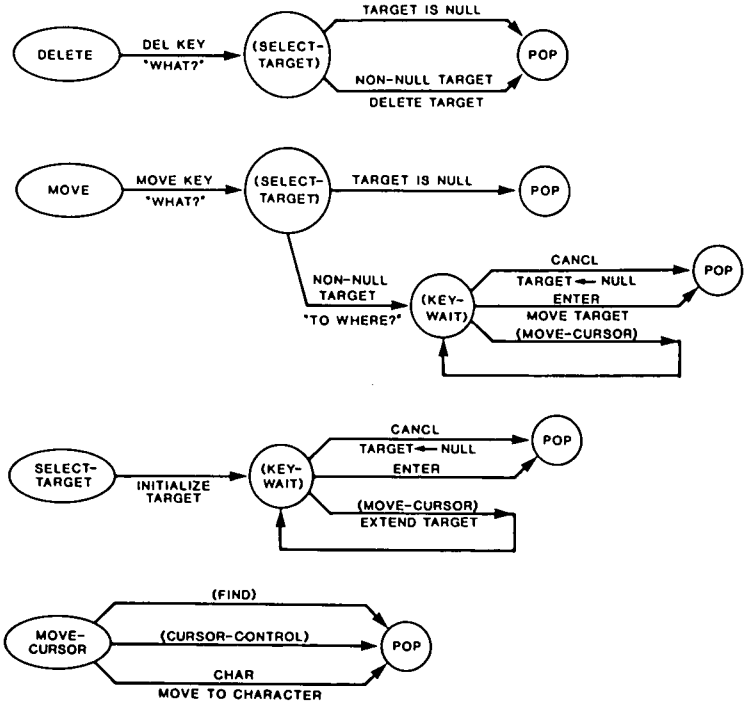

FIG. 6. Simplified GTN for the Displaywriter, DELETE and MOVE functions.

the system from the point of view of the user. The similar patterns of interaction are represented with network subroutines such as INPUT and SELECT-TARGET. As described below, this compact and comprehensive representation of a device will be of considerable value when combined with a representation of the user's knowledge.

3.3. THE RELATION BETWEEN THE TASK AND DEVICE REPRESENTATIONS

An important set of issues concerns the *task-to-device mapping*, the relationship between the characteristics of the user's task and the device being used in the task. As described above, a useful concise characterization of the task is the user's goal structure, which can be derived from the production system representation. An equally concise characterization of the device is a diagram showing the hierarchical arrangement of the major functional components of the device. If the device has been represented by a GTN, this is simply the hierarchy of networks and subnetworks. A preliminary hypothesis is that a good task-to-device mapping is one in which the goal structure graph and the device structure graph correspond. Tentatively, correspondence between these graphs means that the goal structure graph has to be isomorphic to some subgraph of the device structure.

The left hand side of Fig. 7 shows the device structure graph for the DELETE function. This graph was abstracted from the device GTN presented above, simply by diagramming the hierarchy of network embeddings. On the right, Fig. 7 shows the goal structure for the whole task, and for the general delete function presented above

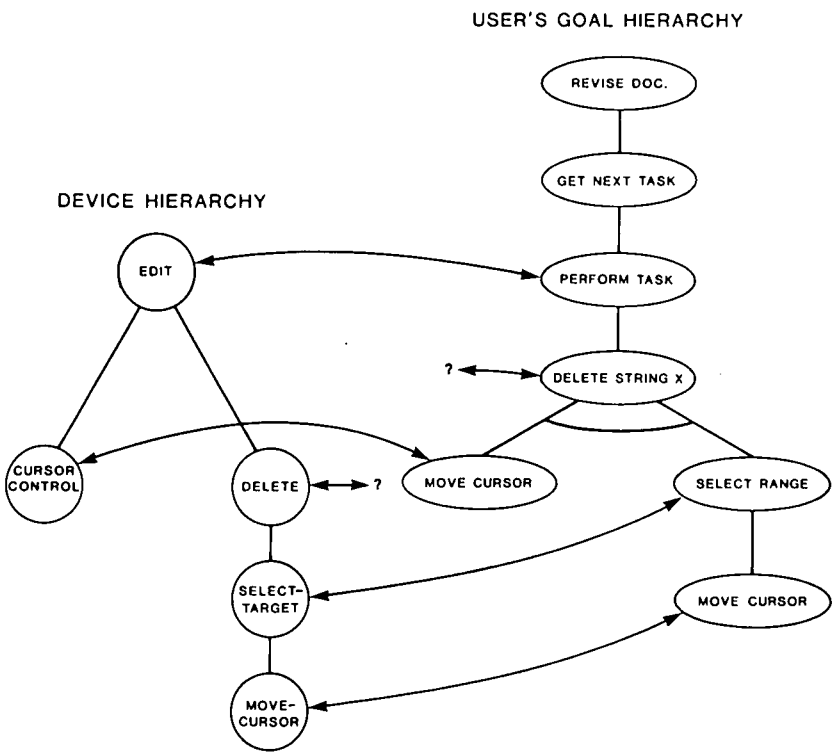

FIG. 7. Example of a bad task-to-device mapping.

(Tables 6 and 7), which was based on the Displaywriter documentation. The curved arrows place the nodes of the two structures into correspondence. Notice that the two structures do not contain isomorphic subgraphs. That is, the goal DELETE STRING X does not map onto the portion of the device structure that one enters upon pressing the DELETE key. Thus, if one's goal is to delete a string, the first subgoal is to move the cursor to the beginning of the string. Then one can press the DELETE key to enter the delete function portion of the device. This means that the pressing of the DELETE key does not correspond to the assertion of a deletion goal. Instead, pressing this key must be deferred until the cursor positioning is accomplished. We would predict then, that there could be many errors associated with confusing the order of execution of the first cursor movement and the pressing of the DELETE key. A similar problem arises for the MOVE function.

The problem could be solved either by changing the design of the device, or by changing the user's goal structure. Figure 8 shows a good task-to-device mapping,

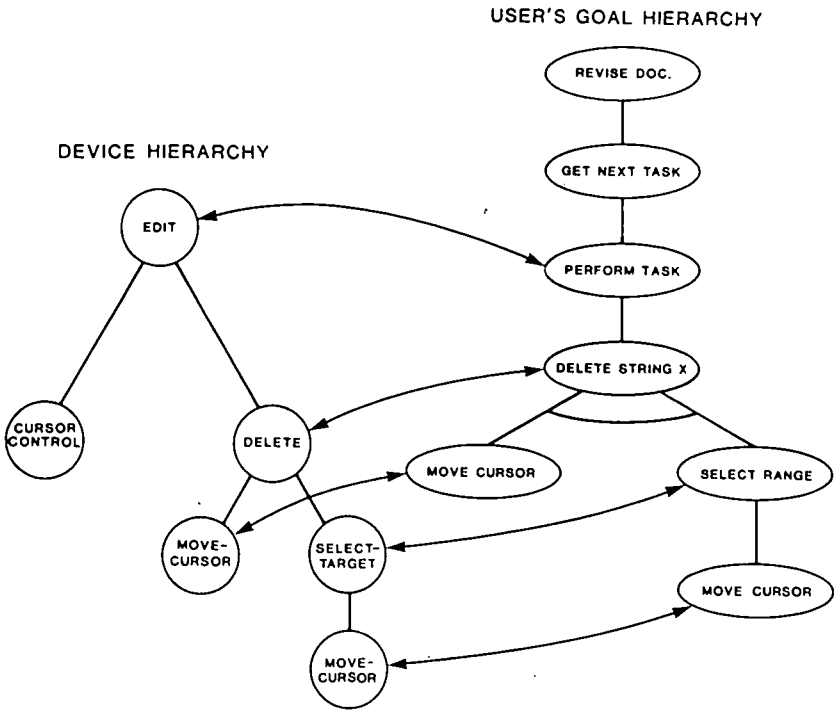

FIG. 8. A good task-device mapping using the same goal structure as in Fig. 7, but a redesigned device.

obtained by redesigning the device and leaving the goal structure the same. The change in the device is that when one presses the DELETE key, one is first prompted to place the cursor at the beginning of the string, and then prompted to place the cursor at the end of the string. The nesting of the device structure is then isomorphic to the hierarchy of goals. In effect, what this does is to ensure that the user progresses through the goal structure, in step with the structure of the device. As soon as one asserts a delete goal,

one can enter the deletion function of the device. Thus, the device's prompts always correspond to the user's current goals, meaning that the device is helping the user to keep track of what he or she is doing.

The alternative solution is to redefine the user's goal structure. This is not necessarily difficult, if it is assumed that the documentation is responsible for giving the user a goal structure. However, in the case of the documentation for the Displaywriter, this appears to be done implicitly, rather than explicitly. Apparently, the underlying assumption of the design of the Displaywriter is that the user will position the cursor at the site of the next unit task *before* spending any mental effort on the nature of the unit task to be accomplished. This goal structure is implied by many screen editors, and assumed in the analysis of an editor presented by Card *et al.* (1980, 1983). However, the documentation for the Displaywriter is organized by the nature of the unit tasks, and not by the overall task strategy. For example, it specifies that to delete a word, position the cursor, then type the DELETE key, and so forth. It does *not* say that to do an edit, first position the cursor to the beginning of the site, then examine what the change is, and finally press the corresponding key.

Figure 9 presents the original device structure and a goal structure that specifies that the positioning operation is always done before the deletion goal is asserted. Notice that in this case, the user's goal structure is isomorphic with the original device structure. So, again, the user and the device will be proceeding through their structures in synchrony; this means that the device is helping the user to keep track of the goals, rather than requiring extra memory load.

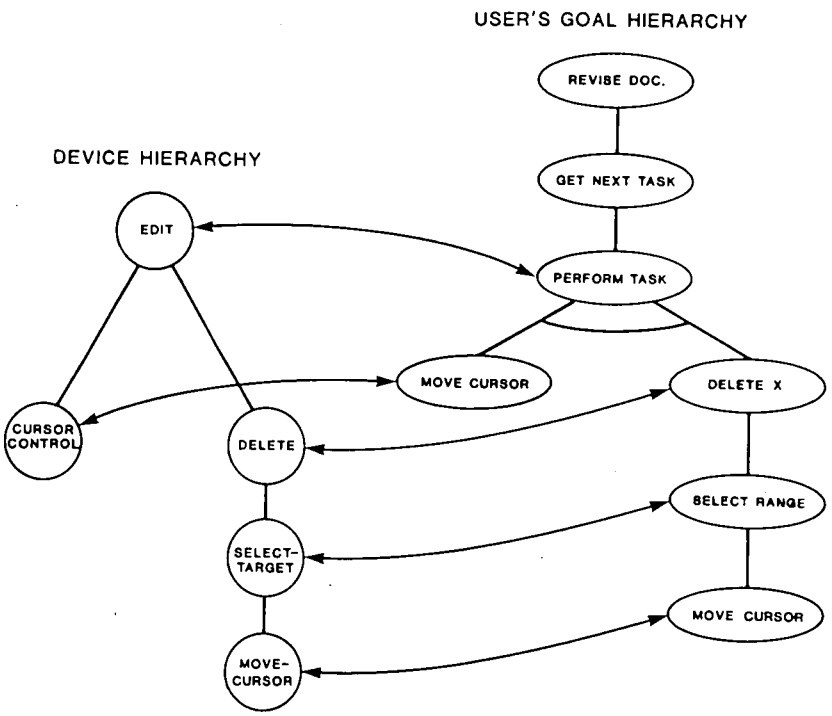

FIG. 9. A good task-device mapping using the same device structure as in Fig. 7, but a different user's goal hierarchy, based on different training.

This example of task-device correspondence shows that a design for a device is not necessarily bad in itself; it depends on the goal structure that the user is taught to use, or might already have. Given a device, the appropriate goal structure can be specified, and then used to organize the documentation. Given a goal structure based on an analysis of an existing task environment, the structure of the device that would correspond to the goal structure can be specified.

It should be pointed out that the above comments on the effects of good and bad task-to-device mappings are speculative. It will take an appropriate program of empirical research to verify and correct these hypotheses.

4. Conclusions

4.1. SUMMARY OF THE REPRESENTATIONS

This paper has presented three representation systems: the user's job-task representation, the representation of the device, and the user's device model. The properties and relationships between these representations can be summarized as follows:

The user's job-task representation describes the person's understanding of when, and how, to carry out the various tasks that make up the job environment. This representation includes the knowledge necessary to decide when various tasks should be performed and the goals, operations, methods, and selection rules that are required to perform each task. The basis structure of a task is characterized by the hierarchy of goals and subgoals that are set up and fulfilled during the performance of a task. This goal structure can be extracted from the job-task representation, and used as a concise description of the structure of a task from the user's point of view.

The user's model of a device is a representation of a person's understanding of the device. Effective user models should provide explanations for the goals and methods appearing in the job-task representation that are specific to the device. Successful explanations link these goals and methods to the user's existing knowledge by using analogies or very simplified explanations of the actual structure and function of the device.

The formal model of the device is a representation of the behaviour of the actual hardware and software, which can be placed in correspondence with the user's job-task representation. It is necessary to have such a representation in order to arrive at an accurate representation of the user's job-task knowledge or device model. It also makes possible the formal description of the relation between job-task knowledge and the actual device.

4.2. THE NEED FOR SIMULATION MODELS

The device and job-task formalisms proposed in this paper have been chosen for their ease of use in the computer simulation techniques used in the development of cognitive theories (Newell & Simon, 1972; Anderson, 1976; Kieras, 1982*a*). Actually setting up and using simulation models based on these formalisms is necessary to realize the full benefit of a formal analysis of user complexity.

An experimental simulation system for modelling the user-device interaction has been developed along the lines diagrammed in Fig. 10. The user is represented by the production system job-task representation, and the device by a GTN. There are two

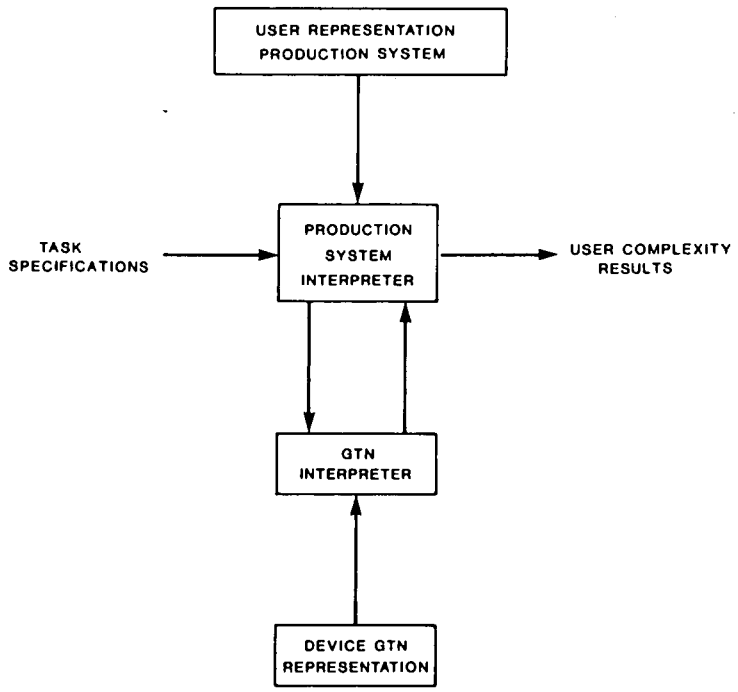

FIG. 10. The structure of the user-device interaction simulator, showing the user and device representations, and the interpreter of each one.

interpreters, one for each representation, that interact to simulate the exchange of signals between the user and the device. The specifications for the tasks to be performed are the input to the user representation interpreter; these specifications would consist, for example, of the coded description of a marked-up manuscript that required editing on a word processor. The outputs of the simulation are the user complexity results collected by the production system interpreter. These are items such as the number of productions fired, goals stacked in working memory, keystrokes executed, and so on. This information will provide the quantifiable measures of user complexity described above. Preliminary work with the simulation system suggests that constructing simulations of user-device interactions for systems as complex as a word processor is indeed practical and useful.

For example, the simulation was used to examine the trade-offs between two different methods for deleting a word. The production system contained the rules shown in Tables 2-7, and additional rules for cursor movement. The device representation consisted of the GTN shown in Fig. 4-6. The task specification consisted of deleting a specified word at a certain location. In one case, the ENTITY specified was a word; in the other case, it was the same actual operation, but the ENTITY was specified as a STRING. Table 8 shows the quantification measures resulting from the simulation.

Both methods involved exactly the same number of keystrokes, and the same peak working memory load, but otherwise there were substantial differences. The general method requires learning and executing more rules, performing more execution cycles, and keeping more items in working memory on the average and for a longer time.

TABLE 8
Comparison of general and specific deletion methods for deleting a word

Measure	Method	
	General	Specific
Number of productions to be learned (including selection and control)	8	4
Number of productions fired	13	9
Number of keystrokes	5	5
Cycles to complete task	12	9
Peak items in memory	6	6
Average items in memory	4·3	3·8
Integrated items in memory (number of items over time)	52	34

However, to replace the general method would require several specific methods. Thus, we see that there are trade-offs between the two approaches, which do not show up in terms of number of keystrokes, but rather in terms of the amount of procedural knowledge and working memory load involved.

One of the primary reasons for using computer simulation techniques is to demonstrate that a complex psychological model can perform the task that it was built to explain. Due to the complexity of the representations introduced above, computer simulation is, in fact, the most efficient and convenient way to determine whether a proposed representation of a user's procedural knowledge has been prepared correctly. Production systems are like programs; attempting to run them is the best way to identify and correct errors. For example, the initial forms for the rules illustrated in Tables 5, 6, and 7 were incorrectly represented in a fundamental way. Another theoretical motivation for setting up simulations is to be able to rigorously generate predictions to compare with the performance of users. This comparison will show whether a proposed, correctly running job-task representation has the same performance characteristics as human users.

4.3. THE PRACTICAL VALUE OF THE FORMAL ANALYSIS

From a practical perspective, the development of simulation models using the formal representations will provide a powerful design methodology for new devices. This approach would permit designers to develop simulated prototypes of the device, and a specification of the knowledge required to operate it, before going on to the development of actual hardware and software prototypes. Examining the quantitative characteristics of these simulated devices and users would permit the detailed evaluation of the relative complexity of alternative designs of a device.

The quantitative measures of user complexity resulting from simulation modelling would permit a description of the various user complexity trade-offs that are involved in the design of a device. Possible quantitative measures include the number of productions in the complete job-task representation, the maximum number of goals in working memory during the performance of a given function, the number of conditions and actions in a production, and so on. At this point, a detailed understand-

ing of the relationships between various static parameters, such as the number of productions, and dynamic parameters, such as the maximum number of goals and notes in working memory, and user performance is not available. This requires empirical research, which is now in progress in our laboratories.

There are several other uses for the formal representations presented here. They can be used as a basis for the generation and evaluation of training materials and reference documentation, because it is relatively easy to translate the production system notation to and from the type of prose used in instruction manuals. Another use would be to describe the differences between experts and novices in terms of their respective job-task representations. From these analyses, it might be possible to develop training materials that would facilitate the transition from novice to expert.

In summary, the formal representations that have been presented in this paper provide the tools necessary to explore the psychological aspects of the complexity of a device, and to provide the quantitative metrics for user complexity that are necessary for applications of these theoretical ideas in the design of actual products.

References

ANDERSON, J. R. (1976). *Language, Memory, and Thought.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

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

ARBIB, M. A. (1969). *Theories of Abstract Automata.* Englewood Cliffs, New Jersey: Prentice-Hall.

BETKE, F. J., DEAN, W. M., KAISER, P. H., ORT, E., & PESSLIN, F. H. (1981). Improving the usability of programming publications. *IBM Systems Journal*, **23**, 306-320.

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., MORAN, T. P. & NEWELL, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

EMBLEY, D. W. (1978). Empirical and formal language design applied to a unified control construct for interactive computing. *International Journal of Man-Machine Studies*, **10**, 197-216.

FOLEY, J. D. (1980). The structure of interactive command languages. In GUEDJ, R. A., Ed., *Methodology of Interaction.* North-Holland, pp. 227-234.

FOLEY, J. D. & WALLACE, V. L. (1974). The art of natural graphic man-machine conversation. *IEEE Transactions on Software Engineering*, **62**, 462-471.

GENTER, D. & STEVENS, A. L. (Eds) (1983). *Mental Models.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

JACOB, R. J. K. (1982). Using formal specifications in the design of a human-computer interface. *Proceedings of a Conference on Human Factors in Computer System.* Gaithersberg, Maryland.

KARAT, J. (in press). A Model of problem solving with incomplete constraint knowledge. *Cognitive Psychology.*

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

KIERAS, D. E. (1982*b*). What people know about electronic devices: A descriptive study. *Technical Report No. 12*, University of Arizona.

KIERAS, D. E., & POLSON, P. G. (1982). An outline of a theory of the user complexity of devices and systems. *Working Paper No. 1*, University of Arizona and University of Colorado.

KIERAS, D. E. & BOVAIR, S. (1983). The role of a mental model in learning to operate a device. *Technical Report No. 13*, University of Arizona.

MINSKY, M. L. (1967). *Computation: Finite and Infinite Machines.* Englewood Cliffs, New Jersey: Prentice-Hall.

NEWELL, A., & SIMON, H. A. (1972). *Human Problem Solving.* Englewood Cliffs, New Jersey: Prentice-Hall.

NILSON, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence.* New York, New York: McGraw-Hill.

NORMAN, D. A. (1982). Some observations on mental models. To appear in GENTNER, D. and STEVENS, A., Eds, *Mental Models.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

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

REISNER, P. (1981). Formal grammar and human factors design of an interactive graphics system. In *IEEE Transactions on Software Engineering,* **SE-7**, (2), 229-240.

REISNER, P. (1982). Further developments toward using formal grammar as a design tool. *Proceedings of a Conference on Human Factors in Computer System.* Gaithersberg, Maryland.

SIMON, H. A. (1981). *The Sciences of the Artificial.* Cambridge, Massachussetts: MIT Press.

WOODS, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the ACM,* **13**, 591-606.