a window interface, a reasoning system, an adaptive user model (AUM) which relies on coaching knowledge and domain knowledge, and a parser (Fig. 7).

To use a computer language effectively, a student needs to understand its syntax and semantics. So, it is reasonable to use a language definition as part of the structure of the AUM and use the language definition as a way to classify user progress and to guide instruction. This would include the domain knowledge used to compose *statements* and the *token* types used. This definition by itself, however, does not include all knowledge needed to understand and use a computer language, because a user must also understand fundamental programming concepts and relationships. The AUM should represent the *concepts* underlying the language, and the *basis sets* necessary to accomplish defined tasks (see Section 6.3.1). Each *statement*, *token*, *concept* and *basis set* may be referred to as a *learnable unit*, that is, the smallest quantity of information represented as a discrete entity to a user. Each of these *learnable units* is represented in frames with named slots for useful attributes, one frame for each *learnable unit*.

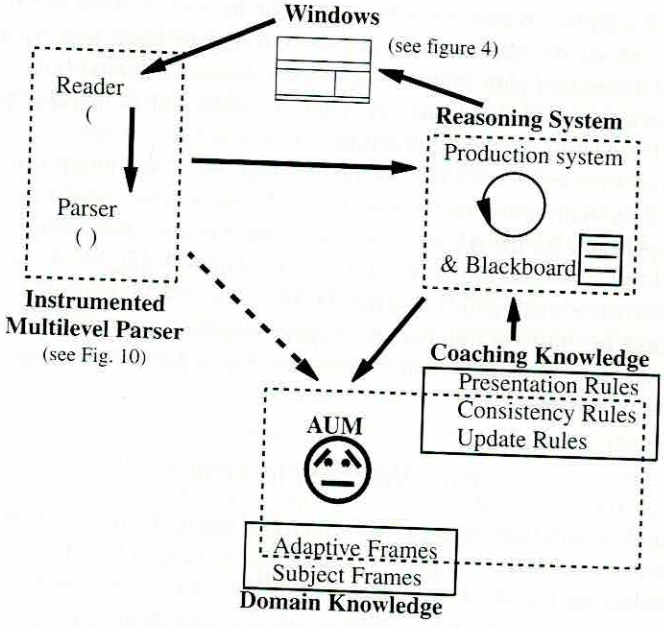

FIG. 7. Dashed lines in the figure represent logical relationships, solid lines represent physical relationships. COACH is composed of interacting parts or objects. The window interface manages text editing, output formatting and menus. The reasoning system creates and uses the AUM to display domain knowledge help and to modify domain knowledge. Coaching knowledge controls these reasoning activities. A multilevel parser notes a user's work context and dispatches information to the reasoning system.

The adaptive automated help architecture must represent examples of each of these *learnable units* and a model of the status of the student in terms of the particular student's understanding and ability to use each one.

A user model frame is recorded for new user-defined *learnable units* as they are created, allowing the system to give help for these as well. A skill domain like Lisp, for which a user is being helped, is represented in the system by these syntactic and conceptual parts. Rules draw on knowledge in frames as they update user help and frame knowledge. A simplified blackboard mechanism allows the knowledge module to propose and veto help text before it is presented. The presentation rules build a list of help items to present. Veto rules and help presentation space constraints eliminate all but the most appropriate items of help.

The AUM relies on the production system to make decisions based on the user model it has built and to decide how to advise the user. The architecture relies on AI technology both for building the AUM and for guiding instruction. The guiding knowledge is embodied in domain knowledge facts and coaching knowledge rules.

Domain knowledge is represented in the help system parser grammar and in subject and adaptive frames (see Section 6.3.1 below). *Subject frames* contain knowledge about the skill domain a user is trying to learn (e.g. Lisp). These frames are associated with each *learnable unit. Adaptive frames* hold usage data and user examples for each function. They are collected as the user works and comprise the AUM knowledge structure.

Coaching knowledge is contained in rules that create and control the adaptive frames and the help presentation blackboard. *Update rules* control the recording of user experience for the AUM. *Consistency rules* contain knowledge about how to build the AUM. These two rule sets work to update the AUM. *Presentation rules* embody knowledge for using the AUM.

The parts or "objects" guided by expert systems knowledge comprise the COACH adaptive automated help architecture. These parts and the way they work together are described in the following sections.

6.1 Window Interface

The window interface manages the screen real estate. It provides separate panes for help text, user input, computer output, and for a menu by which the user can request help (see Fig. 8). It dispatches input key and mouse events to the other modules and presents computer response and advisory help text.

Text-based interactive environments generally type computer output, help, and error messages to a single user console window. The user also types into this window. Confusion often arises concerning which text the computer typed and which the user typed. The combination of such different streams of information

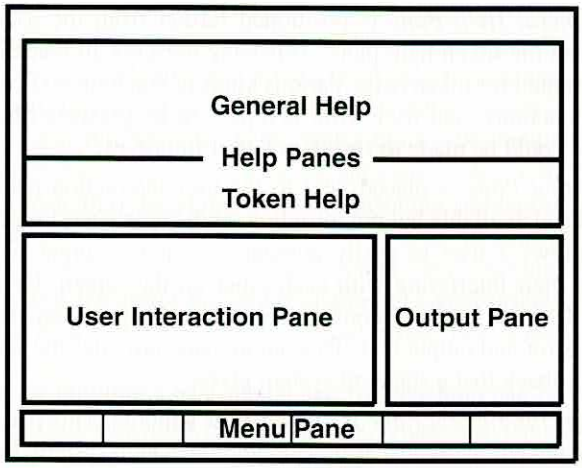

FIG. 8. The window interface separates user input from help and system responses. A menu at the bottom allows a user to request help directly.

into one communication channel requires that the user remember which text on the screen was written by the user and which was written by the computer.

The window interface design used in the initial COACH study physically separates user input from computer output and advisory help. This segmentation insures that computer help and advice do not physically interfere with user input. One way to do this is to vertically separate the token help pane and the general help pane from the user interaction pane and the computer output pane. The user interaction pane does not lock the keyboard when an error intervenes; instead, the character that caused the error (reported on other panes) is highlighted. Using character highlighting to replace keyboard locking and separate panes to preclude typing on the user interaction pane provide visual aids permitting the user to focus more attention on the problem to be solved and less on the computer mechanics.

More specifically:

- The *Token Help Pane* is positioned as closely as possible to the user interaction pane to allow a user to see it easily while typing. This pane is designed to give focused help concerning the specific characters a user is typing. For example, when a novice is typing a token (e.g., a number, symbol, defined variable, etc.), the token help pane displays help concerning that token. When the computer can reasonably predict the next token required, such a pane provides advice concerning it. The pane shows the token name and, as described in Section 6.3.2, presents various levels of description, example and syntax help adapted to the particular user. This local information would not be as useful to users at intermediate and expert levels of proficiency. The adaptive user model chooses when to eliminate this kind of help.

- The *General Help Pane* is positioned farther from the user's immediate view than the token help pane. This pane presents all teaching knowledge not presented for token help. Various kinds of teaching text concerning concepts, functions, and user work compete to be presented here. This help window could be made to shrink as a user improves.
- The *Output Pane* is placed next to the user interaction pane, so as to be noticed and available but not intruding on the user's workspace. This placement allows a user to easily compare computer output with input text, without their interfering with each other on the screen. By contrast, on a standard "Lisp listener" console, user text competes on the screen with system error and output text. This output pane provides the same output and error feedback that a standard system gives.
- The *User Interaction Pane* is a text editor window. This pane allows users to enter and edit work just as they do without a help system. Errors are highlighted in sequence to allow users to correct them in an organized way. As users improve and the need for other help windows diminishes, this window could be made to take up more screen real estate.
- The *Menu Pane* would be a fixed menu area with spatially separated words that call attention to additional system support. This menu pane allows the user to request explicit interaction by using a mouse, rather than by the usual method of typing requests in a console pane and possibly obscuring current work already there. While the help panes ordinarily would automatically provide computer generated assistance to the user, the menu would permit users who recognize the need for help to initiate a request for it. The help generated by the request would appear on the help panes. By selecting menu items with a mouse, the user could ask the computer to give help, show undefined variables or functions and so on. This pane would give the user an explicit medium for interaction with the coaching system. The user would also use the menu for such routine functions as saving and reading files, logging in or logging out.

  Pressing a mouse button in the other window panes could be arranged to provide appropriate so-called "pop up" menus useful for the context of the window.

6.2 Reasoning System

The reasoning system controls all aspects of user monitoring and assistance in COACH. It is made up of a production system which interprets rules and a simplified blackboard which resolves presentation goal conflicts. Together they operate on the adaptive user model (AUM) by referring to domain knowledge and using coaching knowledge to make decisions, as described in the following sections.

In the two decades since MICRO-PLANNER demonstrated the utility of using a rule interpreter, or production system, to achieve reasoning tasks represented in rules (Hewitt, 1972; Sussman *et al.*, 1970), many AI architectures have included them (Davis and Shortliffe, 1977; Barr and Feigenbaum, 1984). For a time it seemed that AI and rule systems were synonymous. The components of a rule system are, in fact, basic to AI representation and search (Winston, 1977; Barr and Feigenbaum, 1984). In order to demonstrate reasoning and learning in real time, COACH limits itself to a forward-chaining rule system. This means that rules are searched through in order and fired when they apply. By breaking the system into small rule sets and not using backward chaining (a goal-oriented rule search), COACH avoids both indeterminate and long searches. This is necessary to allow real-time advising.

Knowledge for building a user model and for coaching can be separated into sets of rules, each of which operates in specific situations. The search time in a rule system can be significantly decreased by breaking reasoning rules into groups or rule sets that are scanned for specific reasoning needs. For example, with this kind of segmentation, if a token is used incorrectly, only rules that provide help for incorrectly used tokens need be consulted. Section 6.3.2 will describe rule sets upon which the model depends for reasoning.

Rules are made up of an antecedent and a consequent. The antecedent part must be true for a consequent part to fire. Both parts are Lisp s-expressions.

Rules may be defined with the following simple Lisp syntax:

```
(DEFINE-RULE
(rule-name rule-set-name)(user-model-parameters)
 IFs-expressions
THENs-expressions )
```

The order of rules in a rule set determines the order in which they will be run. The rule's antecedent, consequent, and position in the rule set determine the reasoning behavior.

A blackboard allows statement-proposals and statement-vetoes to interact in reasoning decisions. Blackboard architectures were first introduced in Hearsay, a speech activated chess playing program (Erman and Lesser, 1975). The Hearsay blackboard was an innovative distributed decision making paradigm for running the system. Different levels of speech recognition each had different parts of a blackboard. Knowledge sources could post proposals and look at proposals on the blackboard. Through this blackboard communication process, multiple knowledge sources collaborated to interpret speech related to chess moves. This architecture has had a continued and marked influence on the artificial intelligence community.

A blackboard is used in COACH to arbitrate adaptive help presentation. Rules in the presentation rule set are knowledge sources that propose and veto various kinds of help to choose the appropriate text to be presented to a user. The order in which

proposals are posted on the blackboard determines their priority. Vetoes might take proposals off the blackboard. A knowledge source decides to present between one and three help text items on the help window. The three highest priority proposals on the blackboard represent text which would then be presented to the user.

6.3 System Knowledge and the Adaptive User Model (AUM)

An Adaptive User Model (AUM) is a formal description of a user relative to a domain that tracks changes in the user's knowledge in that domain. COACH uses an explicit user model. Frames, facts, and rules represent the user and the skill domain the user is learning. The AUM is a set of user model frames (Minsky, 1976) for syntactic and conceptual parts of the domain being coached. While the user is working on a task, these frames record aspects of the user's successes and failures. The AUM for COACH is composed of this representation of the user and an associated reasoning system for creating and accessing knowledge frames. The defined network of relationships between skill domain parts, what the user is doing, and the state of the user model is the basis for selecting user help.

The reasoning system uses this network of domain knowledge and coaching knowledge in the form of rules together with the AUM. Reasoning and planning about how information interacts, the way the system updates the AUM, and even the system's adaptation algorithms reside in rules in COACH. By changing these rules, a researcher could tailor help for different skills and pedagogical theories.

Each skill domain part has a help knowledge frame. These frames can include descriptions, syntax, and example help at the four levels of help proposed by the taxonomy described in Section 5.3.

6.3.1 Domain Knowledge

(1) *Adaptive frames.* The AUM frames for each *learnable unit* have the following user model characteristics or slots:
   (a) *User examples.* Examples are recorded of user errors and user corrections of those errors. When a user makes a mistake, the system records it. When the user is able to correct the mistake, the system stores that "fix" with the example. If the user later makes a syntactically isomorphic mistake, the system displays the familiar earlier example.
   (b) *Usage data.* The following information is also recorded:
      (i) *experience* (how often a particular *learnable unit* has been used by this user);
      (ii) *latency* (how long since the user has used this *learnable unit*);
      (iii) *slope* (how fast the user is learning or forgetting something);
      (iv) *goodness* (a measure of the user's overall performance with respect to this *learnable unit*).

A demonstration of rules that use these slots is contained in Section 6.3.2.

(2) *Subject frames.* Subject frames (Fig. 9) are made up of a subject definition along with various kinds of related material and description text, syntax text and system-defined examples (by contrast with the user-created examples) for the four levels of help.

The reasoning system described above (see Section 6.2) and the multilevel parser described below (see Section 6.4) rely on the subject frames and adaptive frames to run the user interface. The knowledge exists in slots within the frames.

Subject frame styles are defined for each type of *learnable unit* (statements, tokens, concepts and basis sets). Each *learnable unit* has its own frame. Formally, the *learnable units* are represented as follows:

(a) *Language statements: S.* Statements are learnable units which are defined in a syntax facts table described more fully in Section 6.4. This table is extended by user-defined functions. A simplified definition for `PLUS`, for example, is `([ PLUS * N ])`. In this notation described in more detail later (see Tables 4 and 5), the star, "*", indicates that

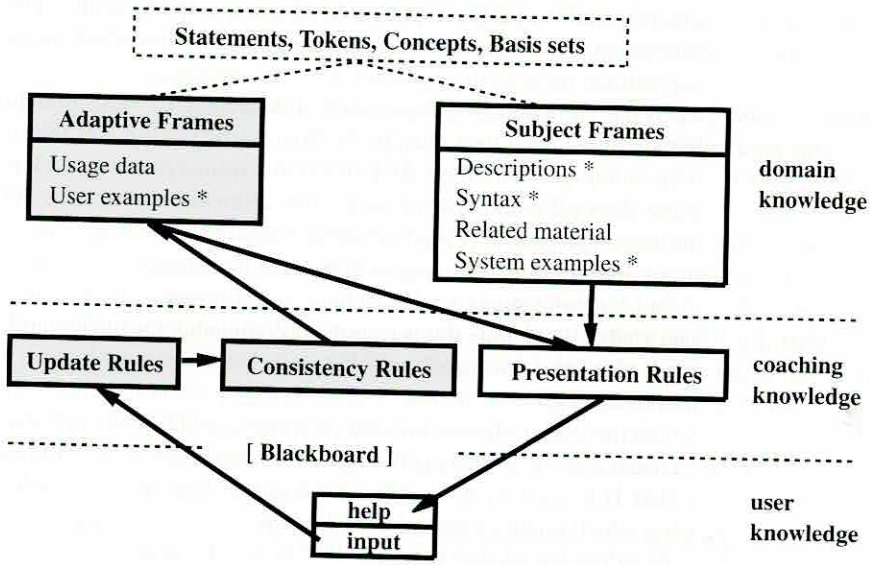

FIG. 9. The knowledge and reasoning structure in an adaptive coaching environment. For each learnable unit, AUM and subject frames are built and controlled by model building and help presentation rules.

what follows can occur zero or any other number of times. Open and closed brackets, "`[`" and "`]`", represent parentheses and the `N` represents a number type argument.

Statement knowledge is a tuple **S ⟨L,SH,D,SE,R,O⟩** where:

(i) *Language syntax: L* is the statement's formal definition which COACH uses to evaluate user work (see Section 6.4.3).

(ii) *Syntax help: SH* frame slots contain formal abstract help definitions of a *learnable unit*. For a specific level in the help taxonomy (starter, model, reference or expert), the syntax describes the *learnable unit* in more detail. Concepts are introduced by examples; the syntax assists a user in internalizing the concepts.

(iii) *Description help: D* frame slots contain help text for each level of the help taxonomy. These slots contain text explaining what a specific *learnable unit* is useful for, information describing when it can be used, and an explanation of what it does.

(iv) *System example: SE* frame slots contain helpful examples typifying a *learnable unit* for a specific level in the help taxonomy. The user examples contained in the adaptive frames described above supplement these system examples.

(v) *Required knowledge: R* frame slots hold the set of skill domain parts with which a user must be familiar to use a particular *learnable unit* (e.g., using the `CONS` function requires a user to understand the evaluation, s-expression, and atom concepts). This set defines a network of related material. Required knowledge makes it possible for a reasoning system to form the strategies needed to create teaching goals in a coaching environment. The required knowledge set details the necessary prerequisites for understanding a particular *learnable unit*. If a user is having trouble with a particular *learnable unit*, COACH displays related things with which the user is already familiar or which could be considered as alternatives. If a user is doing well, this knowledge allows COACH to see how to encourage the user to learn new *learnable units* which relate to ones already known.

Required knowledge is a tuple **R ⟨*F,*C,*T⟩** where:

- *Function: F* is a statement name.
- *Concept: C* is a concept name (described more fully below).
- *Language token: T* is a token type of the domain language (described more fully below).

(vi) *Other related knowledge: O* is a set of *learnable units* which are pertinent to another *learnable unit*. This set defines a network of

relationships which helps the architecture utilize the concepts that tie the domain together. This network is crucial to the coaching that will expand a user's breadth and, when necessary, search for alternative teaching approaches.

Other related knowledge is a tuple **O** ⟨***F, *C, *T**⟩. **F**, **C**, and **T** are defined above in "required knowledge".

(b) *Language tokens: T.* Tokens are *learnable units* which are keywords and acceptable variable types for a skill domain (e.g., "(" , "**CONS**"). They are defined in a table with associated token methods described in Section 6.4.1.

Token knowledge is a tuple **T** ⟨**SH, D, SE, O**⟩. **SH**, **D**, **SE**, and **O** are defined above in "language statements".

(c) *Concepts: C.* Concepts are *learnable units* which are semantic ideas not codified by syntactic parts (e.g., evaluation, iteration, stored variable, etc.).

A *concept* is a tuple **C** ⟨***F, *C, *T**⟩. **F**, **C**, and **T** are defined above in "required knowledge".

(d) *Basis-sets: B.* Basis sets are *learnable units* composed of groups of other *learnable units* comprising minimal sets of knowledge necessary to understand a topic. This term is borrowed from mathematics where it defines a similar concept.

An arithmetic basis set, for example, would require a user to know about `PLUS`, `DIFFERENCE`, and the List and Number concepts. The elements of a basis set are skill domain parts, all of which must be known to do a task in a particular topic area (e.g., the basis set for "simple-lists" includes `CONS`, `CAR`, and `CDR` and the atom and s-expression concepts). Generally, basis sets will be a subset of a required knowledge set; for example required knowledge for `List` includes the Eval concept as well as `CONS`, `CAR`, and `CDR`. Basis sets may be elements in **O** or **R** sets of *learnable units*. These allow the system to reason about basic knowledge a user may be missing when trying to use a *learnable unit*.

A *basis-set* is a tuple **B** ⟨***F, *C, *T**⟩. **F**, **C** and **T** are defined above in "required knowledge".

The subject frames described in this section create a domain representation in COACH. This representation consists of syntax, descriptions, system-examples, and related materials for each *learnable unit* in the domain. The language syntax definitions, **L**, define knowledge with which COACH can record correctness of user work. Required knowledge, related material, and basis sets, included as **R**, **O**, and **B**, define relationships between parts of the domain, much like the links in a hypertext system (Conklin, 1986). This network, **R**, **O**, and **B**, can in fact be browsed like hypertext. More importantly, these are the basis for COACH

reasoning about relationships in the subject domain. Rules like "explore exploration" and "out of practice" described in the next section use these to orient and teach users.

These subject frames are augmented by the AUM to give a rich representation from which coaching knowledge makes decisions.

6.3.2 Coaching Knowledge

Coaching knowledge is embodied in rule sets which suggest information to place on help windows.

Rule sets for creating and maintaining the AUM consist of update rules and consistency rules. These rule sets change the AUM frames each time the parser signals a change in parse state. In this way, user model frames are changed each time a function is closed, a token is typed, a token is found to be undefined, etc.

Presentation rules consult the parser, the AUM and the blackboard to make decisions about what to present. Detailed analysis of two of the more interesting and complex of the presentation rules is provided below.

(1) *Update rules*. A simple update rule set consists of rules with the following mnemonic names:
   (a) *Note-Success* (activated by a correct usage of a *learnable unit*; improves user rating on AUM frame slots for a *learnable unit* and related material).
   (b) *Note-Failure* (activated for an incorrect usage of a *learnable unit*; decreases user rating on AUM frame slots for a *learnable unit* and related material).
   (c) *Was-Bad-but-Getting-Better* (activated when a user has a success with a *learnable unit* which has been problematic; increases user ratings).
   (d) *Was-Good-but-Getting-Worse* (activated when a user begins making mistakes for a *learnable unit* which has previously been rated well; decreases ratings slowly).
   (e) *Best-and-Getting-Better* (activated when a user continues to use a *learnable unit* correctly at more sophisticated ratings; bumps up Best).
   (f) *Worst-and-Getting-Worse* (activated when a user continues making mistakes in use of a *learnable unit* that has been being used poorly; bumps down Worst).

(2) *Consistency rules*. A consistency rule set would work with update rules to create and maintain a user model. A simple consistency rule set has the following mnemonic names:
   (a) *Note-Used* (activated each time a *learnable unit* gets used).
   (b) *Maintain-Best* (works with Best-and-Getting-Better to bump up Best).

(c) *Maintain-Worst* (works with Worst-and-Getting-Worst to bump down Worst).
(d) *Bound-Goodness-and-Best* (activated to record user's "personal Best").
(e) *Bound-Goodness-and-Worst* (activated to record user's "personal Worst").

(3) *Presentation rules*. Presentation rules determine the help that will be provided to the user, posting and removing the various possibilities on the blackboard. Specific presentation rules "argue" for their position and the blackboard records the result which is presented to the user. Separate presentation rules exist for statements, tokens and concepts. Their particular order determines which text will have priority for use as help.

For the purpose of creating COACH (and the evaluation of COACH in Section 8), a model rule set for presentation of statements contains the following rules mnemonically named:

(a) *Losing-Ground* (provides the user with the most basic help).
(b) *Out-of-Practice* (reminds the user of information that was previously understood).
(c) *Encourage-Exploration* (suggests useful information not presently being used).
(d) *Veto-Overly-Sophisticated-Help* (protects the user from help beyond an appropriate level).
(e) *Veto-Extra-Help* (protects the user from too much help).

To give a feeling for how these rules work, an examination of a few of them follows.

A simple "Losing-Ground" rule provides a user with examples:

```
IF
   learnable unit used has a low Goodness score
   and a low learning Slope,
THEN
   PUSH a User-Example onto the blackboard, and
   PUSH a System-Example onto the blackboard.
```

This rule implements the following concept: if the Slope and Goodness measure are both low, then the person is doing poorly. In such a situation, the rule proposes that both a prior user-example, if available, and a system-example of correct use of the statement be placed on the blackboard for the confused user.

Besides pushing things onto the blackboard, the system might use other rules, such as "Veto-Extra-Help", to take inappropriate information off the blackboard.

```
IF
   user expertise for this learnable unit is NOT
   better than the Best it has been
```

```
THEN
  PUSH Veto-Extra-Help onto the blackboard.
```

This rule implements the following concept: if a user's expertise is not at its highest point so far, tell the blackboard not to provide overly verbose help.

The defined network of relationships is used in rules such as "Encourage-Exploration" to expose a user to new information.

```
IF
  learnable unit has a high Goodness score, a
  non-negative Slope, and has been used many Times
THEN
  PUSH previously unused Related and Required
  knowledge for the learnable unit onto the
  blackboard.
```

This rule implements a tutoring concept: if a user is doing well with a learnable unit, expose them to more material in that area of knowledge.

6.4 Instrumented Multilevel Parser

The domain a user is trying to learn (e.g., Lisp) has a syntax—the set of things a user can type that are correct and interpretable. Like the standard UNIX facility, Yet Another Compiler Compiler (YACC), and LEX (Kernighan and Pike, 1984a), COACH includes a general-purpose parser which uses state machines to classify character and token types to drive lexical analysis. A formal language definition drives actual parsing strategy. Unlike other parsers, the COACH parser is *instrumented* to run rules and add knowledge to a user model after each keystroke.

The multilevel parser (see Fig. 10) structurally separates different kinds of data about the user's interaction with the system. It is made up of a character classifying table, a token parse table, a token attribute grammar parser, and a statement attribute grammar parser. The parser structure, function, and syntax are described below.

6.4.1 Parser Structure

Character Classifying Table. The character classifying table lists all the characters and their functions. It is an efficient mechanism for classifying each character's impact on a user's work. Character types can readily signal delimiters and user mode changes with one computer array reference instruction. Use of this technique for the two "bottom-most" analysis levels is part of the overall strategy of real-time response required to provide an automated adaptive coaching interaction style.

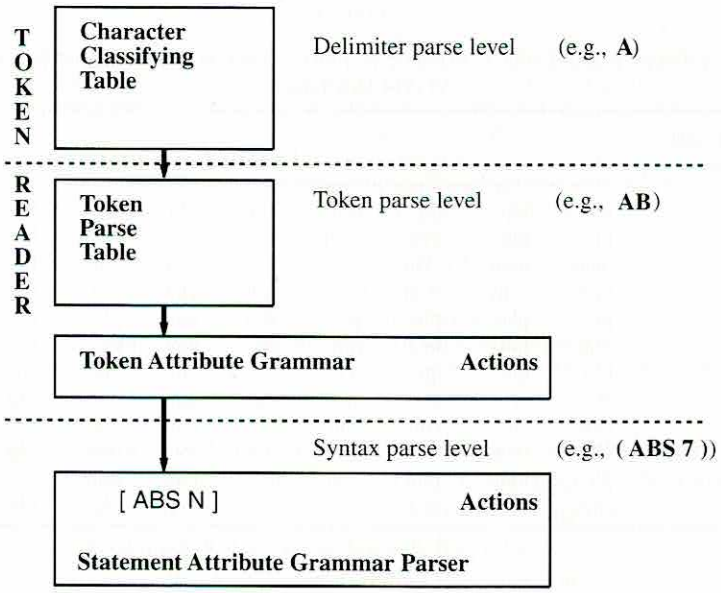

FIG. 10. The structure of a multilevel parser.

Token Parse Table. A token or word in a user input language has meaning by nature of its kind or "type"; it might be a number, keyword, variable name, etc. The token parse table (see Tables I–III) notices token type changes and dispatches tokens to the token reader. It is the next level of user input analysis. As with the character classifying table, the token parse table uses a lookup technique, which permits most user context changes to be recognized without resource intensive reasoning.

Token Attribute Grammar Parser. Recognition of a token could have side effects. The token attribute grammar parser is tied to the token parse table to handle token level interpretation of user typing. Each token has an associated "Object method" which analyzes the impact of a token's completion. The methods call for coaching help, update the user model, and change the way COACH views the domain and the user.

Statement Attribute Grammar Parser. Syntax is the defined order in which tokens can legally follow each other. The statement attribute grammar parser handles the lexicon and builds a structure describing the syntactic unit the user is typing. User input is filtered through this parser for lexical analysis. Static semantics refers to the meaning attainable from a program without running it.

TABLE I

MODEL TOKEN PARSE TABLE SUFFICIENT TO BREAK LISP INPUT INTO DELIMITERS, ERROR STATES AND TOKENS

Current state / Character read	pls	min	sym	qte	st	cmt	num	shp	hlp
er	hlp	hlp	hlp	hlp	hlp	hlp	hlp	hlp	hlp
sp	hlp	hlp	sym	sym	st	cmt	hlp	hlp	hlp
num	num	num	sym	sym	st	cmt	num	hlp	min
chr	sym	sym	sym	sym	st	cmt	sym	hlp	min
opn	pls	pls	pls	pls	st	cmt	pls	hlp	hlp
cl	min	min	min	hlp	st	cmt	min	hlp	min
qte	hlp	qte	qte	qte	st	cmt	qte	shp	hlp
st	st	st	st	st	min	cmt	st	hlp	hlp
cmt	cmt	cmt	cmt	cmt	st	cmt	cmt	hlp	hlp
ecm	pls	min	min	hlp	st	min	min	hlp	hlp
blnk	pls	min	min	min	st	cmt	min	hlp	hlp
shp	shp	shp	sym	sym	st	cmt	hlp	hlp	hlp

TABLE II

SYMBOLS IN A TOKEN PARSE TABLE BREAK UP INPUT INTO POSSIBLE DELIMITERS AND SPECIFIC TOKEN TYPES

er	cntrl characters etc.	Characters not used in the domain.
sp		Special characters.
num	`0 - 9` and `.`	The decimal numbers.
chr	`a - z` and `A - Z`	The standard roman characters.
opn	`(`	Open parenthesis symbol.
cl	`)`	Close parenthesis symbol.
qte	`'`	Single quote is the Lisp QUOTE macro symbol.
st	`"`	Double quote is the string delimiter symbol.
cmt	`;`	Semicolon is the Lisp comment delimiter symbol.
ecm	**lf cr**	Characters that end Lisp comments.
blnk		Blank space character.
shp	`#`	"Pound" or sharp character.

Syntactic and static semantic advisory responses are triggered by this parser. The statement parser sends information about the user to the AUM. As explained below, a simple description language can be used to create statement templates. The parser steps through these templates accepting user input. The parse state pushes onto and pops off a stack as expressions are evaluated.

6.4.2 Parser Function

The mechanics of, and the relationships required between, these elements of

TABLE III

STATES IN TOKEN PARSE TABLE. THESE STATES MODEL AND EXECUTABLE PARSE OF TOKENS. LANGUAGE PARSING IS DRIVEN BY THESE STATES

pls	Is the start of form parse state.
min	Is end of parse context state.
sym	Indicates a symbol is being parsed.
qte	Indicates a quoted object is being parsed.
st	Indicates a string is being parsed.
cmt	Indicates a comment is being parsed.
num	Indicates a number is being parsed.
shp	Indicates a macro is being parsed.
hlp	Indicates an illegal object is being parsed.

the parser will now be described. The token parser can be modeled by a finite state automaton. The COACH interpreter is parsing, not to interpret the domain language, but rather, to build a user model and to note teaching opportunities.

Table I shows details of transitions that can determine Lisp token delimiters for COACH. As characters are accepted by this token parse table, they are added to the partially constructed token. The token parse table takes the current parser "state" (the column) and the current character (the row) as input to determine the reader state. For example, if the reader were in string state, `st`, and received an illegal character, `er`, the reader would change to the help, `hlp`, state.

The character classifying table, the token parse table, and the token attribute grammar parser collectively comprise the token reader. The accepted characters and the state of the parse table drive the token reader.

A function for each token type checks token side effects of the parse state. When a new Lisp variable is read, for example, such a function would add it to the user's environment as necessary.

When a token is accepted, the AUM is updated. A statement is composed of legal tokens in a syntactically legal sequence defined by language parse templates. The statement attribute grammar parser is controlled by language parse templates. Each time a token is accepted by the token reader, the statement parser takes a step through the currently active template and predicts what the user might need to do.

The acceptance of a token and progress through a template cause the rule system to select presentation help. The template state and past input give the AUM knowledge of user goals that are often adequate to predict expected needs (as described in Section 6.3).

The statement parser uses a formal language to describe syntactic parsable expressions in templates. If a statement call is made, a new context is started, for example

```
(SETQ a (CONS
```

makes the CONS statement parse template active, pushing the SETQ parse template onto the pending parse stack. The statement parser steps through these templates, pushing them on and popping them off the pending parse stack, as new contexts are started or completed.

Expression side effects can be further detailed in an action function to be run after a parse is accepted so that COACH can keep a record of the user's environment as well as the user's state. To implement side effects, a token or function can have an action function associated with it. When the adaptive automated help framework has completed recognizing the token or function, the action function will run. The SETQ function, for example, has an action function which adds new variables to the known variables list.

A more complete example can demonstrate the statement parser in action: writing an s-expression to sum three with the product of five and four. Starting in the beginning state, when the user types

```
(
```

the reader puts the statement parser in the pls state. The token rule set now consults the adaptive frames to decide whether help concerning a function name should be displayed for this user, and if so, what kind of help is needed. As the user types

```
PLUS
```

the statement parser makes the PLUS parse template the current template. The function rule set and blackboard now use the AUM to decide how much and what kind of help to present. The token rule set fires to decide what immediate help to present, possibly indicating that a number is called for in the PLUS parse template. If the token rule set demonstrates user need, number help is given to the user as three is typed in. While the product is being entered, the parse stack has to remember the PLUS parse state. The TIMES parse template is now put on the parser stack. The function help rule set is fired again to reason as to what help to present for the TIMES function. When the product is ended with a

```
)
```

the sum parse template comes into force. When the sum parse template is closed, the beginning state would be in force. A rule set now gives top level help if appropriate.

Each transition change above causes an action function, if it exists, to run when a token or function is recognized.

The multi-level parser rule system and the AUM work together as an architecture for adaptive help.

6.4.3 COACH Syntax

Languages can be formally defined. Specifically, they are defined by a grammar and alphabet (Hopcroft and Ullman, 1979). COACH uses a formal

language representation of a subject domain to assess user work and progress. The domain language is defined for the statement attribute grammar in a formal, context-sensitive syntax notation. This notation is shown with the Lisp system key symbols in Table IV.

Logical conjunction in a template is indicated by juxtaposition. A legal statement or sentence, S, consists of a string of symbols from the alphabet (described in Table V) that satisfies an expression in a legal syntax table.

In the formal language definition, key symbols, control symbols and delimiters are surrounded by a set of parentheses:

```
{S := ( *{A} )}
```

The language can be described as being made up of the alphabet:

```
A := A,S,N,L,F,X,Q,FS,{,},[,],V,?,*,@XXXX
```

TABLE IV

LANGUAGE PARSE MODEL: TOKEN TYPES

A	atom
S	defined symbol
N	number
L	list
F	function
X	any of the above types
Q	check only parenthesis level
FS	function specification

Notes: These allow modeling of Lisp's major token types. Such a parser table is designed to analyze user proficiency; a language parser designed to implement a language might have more types.

TABLE V

LANGUAGE PARSE MODEL SYNTAX DELIMITERS

{	open a syntactic parse unit
}	close a syntactic parse unit
[	(, open a clause
]	), close a clause
*	next syntax part can occur 0 or more times
?	next syntax part can occur 0 or 1 time
V	at least one component of the next clause must occur at least once
@	consider the following characters a symbol

Notes: The parse modeling language itself has immutable token type control symbols to allow designers to describe a language to be coached.

where XXXX is any string of characters. Delimiters should always come in pairs:

```
{}, [].
```

In the Lisp language, a new S is signaled by an open parenthesis, represented by a [, so all s-expressions end with a closed parenthesis, represented by a]. (The UNIX command language ends commands with a carriage return, and so does not require this].)

The COACH architecture uses an attribute grammar parser. Each token type and each completed parse sends an `:action` message when recognized. Each key symbol in the notation has a method (function) associated with it which can cause an action during the parse.

Functions that the system parses are described in this notation. The simple example

```
[ ABS N ]
```

defines the absolute value function as requiring a parameter of type number. A slightly more complicated syntax such as

```
[ SETQ * { A X } ]
```

requires 0 or more atom-anything pairs for a legal "sentence".

6.5 Conclusion

Section 6 has described the COACH architecture. Several representations work together to create help for the user: the subject frames (definitions of the domain), the adaptive frames (recording of a user relative to a domain), the presentation rule sets (which embody a model of teaching), and the multi-level parser (syntax domain definition).

The section has further shown how formal representations are used for domain knowledge, teaching knowledge, adaptive strategy, and the way the adaptive frames are used to create a coaching advisor. The next section discusses additional reasons for using these multiple interacting representations.

The COACH/2 system is somewhat different. It does not require the author to write syntax text. COACH/2 keywords are the spatial and graphical elements on the screen. It defines arguments to these icons, dialog boxes, etc. as the mouse actions or typing that is transmitted to the GUI element.

7. A COACH Shell

COgnitive Adaptive Computer Help (COACH) is a proof by demonstration of a real-time, adaptive, advisory agent. Demonstrations of possibility are important,

but further progress in a field requires tools to make experiments feasible. As well as being a demonstration, COACH was designed to be a testbed for understanding adaptive user interaction. The structure is organized to allow a courseware designer to change the skill domain information, the presentation approach or the adaptive strategy with minimal effort. It has been used to show that COACH can work for open systems (see Section 5) and with different domains, and can support experimentation with user modeling and help presentation strategies.

7.1 Using COACH in Open Systems

Open systems are systems which are too big to be analyzed or which grow with use (see Chapter 5). The Lisp COACH demonstration shows that adaptive user help can be used for open systems.

COACH creates user models and provides help for any number of system functions and new user functions. The system was tested on a twenty-five thousand function Lisp programming environment. Since help text could not be provided for all of the constantly changing Genera Lisp functions, a mechanism for adding help for functions as they get used was provided instead. Multi-level help was provided for a basic set of functions and was automatically augmented to include any other functions actually used or added by a user.

COACH queries the Genera Lisp environment for a syntax description which it uses to start a user model for a previously unknown function. When a user defines a new function, COACH records its syntax to start a user model for this function. As newly added functions get used, examples of correct and incorrect usage are collected for use as example help text.

7.2 Using COACH For Different Domains

COACH works from a courseware designer's definition of a domain language. A UNIX version was created to demonstrate that COACH can be ported to different domains.

Matt Schoenblum, a talented seventeen-year-old high school student without programming experience, was able to demonstrate this capability by adapting the COACH system to teach the UNIX operating system's shell command language (Kernighan and Pike, 1984b) in a ten-week internship. Schoenblum learned enough UNIX to be comfortable using it to edit documents, send mail, transfer data, print things, etc., to accomplish his work. He interviewed UNIX users to identify twenty key UNIX commands and wrote multi-level help text for these commands. He then defined delimiter and token types needed for the system to parse UNIX commands. Finally, he wrote syntax definitions for all identified commands. COACH enabled such an accomplishment by only requiring data, rather than reprogramming, to create a help system for a new domain.

Two functions were provided to handle the new tokens which were used in UNIX but not in Lisp: the carriage return delimiter and the "anything" (or *) token. Changes to the parse table—altering the end-of-statement delimiter from ")" to carriage return and eliminating the ";" for comments—were provided as well. At the time, the COACH system only ran on Symbolics computers. A command caller which interfaced to a UNIX workstation over a Telnet (Kernighan and Pike, 1984b) connection was proposed but has not yet been tested.

This UNIX help system was experimented with by several people, and improved through iterative experimentation. No formal study has yet been performed with it.

7.3 Experimentation With Help Presentation Strategies

COACH also supports experimentation with help presentation strategies. All help presentation is managed by rules. These rule sets have allowed continued testing and changing of the coaching strategies in the COACH implementation (see Chapter 6 above). Several students have experimented with the rule system to learn about adaptive strategies (Matt Kamerman, Kevin Goroway, Frank Linton, and Chris Frye).

These experiences demonstrated that the COACH can be used as a shell for developing proactive, interactive adaptive computer help systems. The COACH implementation gives proof by demonstration that adaptive computer help works in open systems, for different domains, and supports experimentation with adaptation and help strategy. The next section describes experiments which show COACH can improve student performance.

The COACH/2 system has allowed instruction development professionals with no programming expertise to create help content for OS/2.

8. Evaluation of COACH Adaptive User Help

Two user studies have been performed to evaluate COACH. A preliminary study investigated COACH user perception differences. The second study quantitatively demonstrated these differences and performance improvements as well. In this five-session Lisp course, the system was found to improve both performance and perceived usability when compared to a version which offered only non-adaptive user-requested help. This is the first demonstration of an adaptive interface showing performance differences for users.

Enhanced interface features available to both groups may have improved productivity as well (e.g., the pointing device, on-line selectable help, separate input and output windows, real-time error detection, etc.).

8.1 Pilot COACH Study

A pilot study was conducted to evaluate automated adaptive help in the COACH system and to flesh out issues for the full scale study. Six programmers who had no knowledge of Lisp were recruited from research staff, programmers and co-op students at IBM T. J. Watson Research Center. The three day course consisted of a classroom lecture each evening followed by a work period. The students worked through exercise sets, and responded to interview questions and quizzes. Each evening involved a new exercise set. One group performed work using COACH, the other group used a standard interpreted Lisp reader on an IBM PC-RT computer.

Many differences between the actions and reactions of the two groups were noticeable. Students appeared much more energetic and productive when they were using COACH. They used the on-screen help and they put their fingers on the screen. While the COACH students tended to experiment within the system, the other group tended to write ideas on paper. When, on the last night, the groups switched places, the behavior also switched.

However, technical difficulties and the small number of participants make formal analysis of the pilot study uninteresting.

8.2 Quantitative Study; Demonstrating COACH Usability Improvements

Major improvements to the pilot study were included in the full, quantitative study:

- The lecture format of the pilot study was changed to a self-paced format in the full study because it had appeared that the lectures made the students feel pressured. They seemed to believe that the difficulties they were having were caused by the lecture, when in fact, the course was designed to be difficult.
- The students in the pilot study appeared anxious and self-recriminating when they could not finish the entire exercise set provided for that evening. So, for the full study, the three exercises from the pilot study were combined into one unbroken problem set. This relieved some of the unnecessary performance pressure.
- In the pilot study, the group using the adaptive automated help seemed to be enjoying themselves, while the other group did not, so a daily comment sheet was added to the course to record the way students felt.
- Recorded audio interviews were added for the same reason.
- Improvements in the COACH implementation reliability made the mechanics less daunting.

- A version of COACH without an AUM was arranged for the control group. This allowed the study to concentrate on the value of an AUM, the central COACH technology, rather than other ergonomic advantages of COACH.

The full study tested the hypothesis that an adaptive coaching paradigm can improve user productivity. Normally COACH adapts to its user and automatically offers help at an appropriate level of understanding for that user. A method was devised to focus the user study on the comparison of automatic adaptive help with user-requested help. A control version of COACH was created which included all interface aids, but excluded the AUM, which had the effect of eliminating the automated adaptive help. It still separated user actions on the window panes from system actions when reporting errors and displaying user-requested help (Fig. 8). The study compared user experiences with this control version and experiences with the automated adaptive version.

8.2.1 Method

Nineteen employees of IBM T. J. Watson Research Center were recruited. They varied from summer interns to professional programmers. While all of these "students" had prior programming experience, none had previous experience with Lisp.

The students were recruited with an electronic poster. The poster solicited people who knew how to program but had no exposure to the Lisp programming language, and who wanted to participate in a short class/study teaching Lisp. Incentives to participate were sandwich dinners, exposure to the experimental system, and the promise of learning a new language.

The students were separated into an early session meeting from 5:00 p.m. to 6:00 p.m., and a late session meeting from 6:00 p.m. to 7:00 p.m., each day for five days. Attempts were made to assign students to whichever session best fitted their schedules. Students were assigned at random to use the manual help or to use the adaptive help. By the time the course was underway, eight students were using the manual help system and eleven students were using the automatic adaptive help system.

Courseware created to support the user study include a course introduction, a Lisp tutorial, and an EMACS editor reference card. Materials used to evaluate the students consisted of a test given before the course began (pre-test), daily comment sheets and a test given at the conclusion of the course (post-test). In addition, audio interviews and student exercise solutions were used as sources of data. These are described in more detail below.

Pre-Test. Before the course began, the students were tested for their knowledge of Lisp and programming concepts in general. The written pre-test was administered to them to collect background information and to insure that they

had no working knowledge of Lisp. The test included questions to evaluate previous programming experience, to establish which programming languages the students knew, and to measure awareness of common programming concepts. Lisp-specific questions were asked to eliminate any students who had prior Lisp experience.

Precourse Materials. In this study, students worked with COACH on Symbolics workstations with a screen layout as shown in Fig. 8 in Section 6 above. The students were isolated from one another and not permitted to converse with each other. Some had separate offices; one group of three sat facing three different walls in a large office. All users in this "communal room" were members of the non-adaptive group.

Students were instructed in basic operations they would need to use, such as where the rubout key was on the keyboard and how to use the mouse. They were all given the same set of course materials. The materials consisted of a brief Lisp tutorial, a quick reference sheet for the EMACS editor, and an exercise set.

Students were encouraged to use the computer help whenever possible. The tutorial was stapled to the exercises facing backwards to force them to turn over the tutorial to see the exercises.

The students were told they would only receive help from the experimenters in the case of machine problems, not in learning Lisp.

They were instructed to read as little of the tutorial as they felt necessary to familiarize themselves with Lisp, to work on the problem sets in a self-paced manner, and to refer to the help windows often.

Lisp Tutorial. A tutorial presentation of the major Lisp constructs was provided on paper. It was written so as to not to solve the exercise problems, yet complete enough to aid the students with learning Lisp.

A motivational introduction listed advantages of Lisp as an application development language. The format of the tutorial was similar to a textbook; concepts were explained in an order so as to build on each other. Once a concept was explained it was followed by simple examples.

The brief nine-page tutorial covered the range of topics a full Lisp course would cover, which might be daunting to beginning students. They could have read all of this material, but were not required to. If they read it all, they would certainly have been introduced to many more challenging topics than could normally be mastered in five hours.

The topics of the tutorial were:

- What is Lisp, and why use it?
- Read-Eval-Print loop.
- Functions.

had no working knowledge of Lisp. The test included questions to evaluate previous programming experience, to establish which programming languages the students knew, and to measure awareness of common programming concepts. Lisp-specific questions were asked to eliminate any students who had prior Lisp experience.

Precourse Materials. In this study, students worked with COACH on Symbolics workstations with a screen layout as shown in Fig. 8 in Section 6 above. The students were isolated from one another and not permitted to converse with each other. Some had separate offices; one group of three sat facing three different walls in a large office. All users in this "communal room" were members of the non-adaptive group.

Students were instructed in basic operations they would need to use, such as where the rubout key was on the keyboard and how to use the mouse. They were all given the same set of course materials. The materials consisted of a brief Lisp tutorial, a quick reference sheet for the EMACS editor, and an exercise set.

Students were encouraged to use the computer help whenever possible. The tutorial was stapled to the exercises facing backwards to force them to turn over the tutorial to see the exercises.

The students were told they would only receive help from the experimenters in the case of machine problems, not in learning Lisp.

They were instructed to read as little of the tutorial as they felt necessary to familiarize themselves with Lisp, to work on the problem sets in a self-paced manner, and to refer to the help windows often.

Lisp Tutorial. A tutorial presentation of the major Lisp constructs was provided on paper. It was written so as to not to solve the exercise problems, yet complete enough to aid the students with learning Lisp.

A motivational introduction listed advantages of Lisp as an application development language. The format of the tutorial was similar to a textbook; concepts were explained in an order so as to build on each other. Once a concept was explained it was followed by simple examples.

The brief nine-page tutorial covered the range of topics a full Lisp course would cover, which might be daunting to beginning students. They could have read all of this material, but were not required to. If they read it all, they would certainly have been introduced to many more challenging topics than could normally be mastered in five hours.

The topics of the tutorial were:

- What is Lisp, and why use it?
- Read-Eval-Print loop.
- Functions.

- Lists.
- Conditionals.
- MAP & LAMBDA.
- Defining functions.
- Repeated computation; Iteration.
- Repeated computation; Recursion.
- Data structures; property lists.
- Data structures; DEFSTRUCT.

Exercise Sets. The exercise sets contained problems covering basic arithmetic operations, list operations, conditional execution, and a small database project. To avoid a ceiling effect (all or many users completing all exercises), the exercise sets were intentionally longer than what a student could complete in the time available. Solutions to the first nine exercises consisted of expressions composed of built-in Lisp functions. Correct solutions indicated completion of exercises. When the students finished these single-answer questions at their own pace, they began the database project. The students were not told which Lisp functions they should use, nor how they should construct the database.

Arithmetic problems introduced the concept of evaluation order, forcing the students to understand that function names come first in Lisp s-expressions. For example, one student, while trying to accomplish

```
(times (plus 5 5) (plus 2 2))
```

was observed trying

```
((plus 5 5) times (plus 2 2))
```

but was able to understand the problem with the aid of COACH. The system instantly recognized that an error was being made and popped up an attention-grabbing error message in the immediate help window. When this error message was noticed, the student was able to use the COACH on-line help to figure out the problem in the model of evaluation used to construct the statement. The student was able to understand the order of evaluation problem and fix the operator/argument order in the solution.

The list operations required for solutions to the exercises included CAR, CDR and CONS, the concept of nested lists, and QUOTE. The students were asked to create lists of varying degrees of difficulty. The simplest was a single level list (1 2 3), and the most difficult involved a multiple level list that required an understanding of quoting.

A conditional statement was required to solve one of the exercises. The task required the student to cause the computer to print yes if a certain element was contained in a list. Students could have used COND and iteration or recursion to

solve this problem. An easier approach using the MEMBER function could simplify the solution.

Unlike earlier problems, students had to write their own functions in solutions to the database project exercises. One might expect students to use the simplest data structure, a list, or more rarely, property lists, the first time they need accessor functions. The tutorial covered these data structures, and the sophisticated DEFSTRUCT macro as well. Surprisingly, most students chose to use DEFSTRUCT instead of lists or property lists, all of which were covered in the Lisp tutorial. The reason most students gave for this choice of using DEFSTRUCT was, "It does everything for you." This indicates that they understood the value of a macro that writes functions the student would normally have to write.

The database problem was worded carefully to segment the solution into six small user-written functions. For example, the students were asked to write a function to add a person to the database, and to write a function to retrieve a person's phone number from the database. This allowed a direct measure of productivity by the number of functions written.

Analysis of student solutions was used to compare productivity of the two groups. In addition, quality of user solutions was evaluated. Students' code was examined for appropriateness and sophistication of Lisp functions used, use of variables and condition checking as well as overall style.

Comment Sheets. During each of the five one-hour sessions, the students were given a comment sheet to record impressions. They were instructed to write as much or as little as they chose. The following questions were asked on the comment sheet:

(1) How often do you look at the help screen while solving a problem?
(2) How helpful is the help screen?
(3) How helpful is the COACH window system, as compared to a line-based interpreted environment?
(4) Observations about COACH? (Answers from this question were evaluated for perceived value or rating of the COACH environment.)
(5) Observations about Lisp? (Answers from this question were evaluated for perceived utility of the Lisp programming language.)
(6) What problems are you having?
(7) What problem are you working on?
(8) What is your motivation to learn Lisp? (Answers to this problem were given on a scale of one to ten.)

To compare the answers of the two groups, all the written answers were analyzed and coded as varying from zero to five. Two readers evaluated each answer and assigned it a value without knowing which group it came from. These

numerical values were used to evaluate the likelihood that the two groups had the same experience with the system (see Table VI and Fig. 11).

Interviews. Near the end of the course, six randomly chosen students from each group were interviewed for a few minutes on audio tape while they were working. The important question asked was:

- What do you find most helpful while solving a problem: the help screen, the selectable menus, or the tutorial?

Post-Test. At the end of the user study, the students were given a post test. To measure the amount of Lisp learned by each student, questions covered the same material as the pre-test. In addition, questions about specific feelings toward COACH and Lisp were posed. Although similar to the comment sheets, these questions were worded differently to reveal more about the students' personal views.

8.2.2 Experimental Data and Analysis

The following sections give detailed descriptions and analyses of data taken in the study. The data recorded came from the pre-test, comment sheets from each day, saved exercise solutions from each student, verbal interviews of students during their sessions, and the post-test taken after the course was completed.

Pre-Test. The pre-test showed that all students selected were experienced programmers, but had no prior Lisp experience. Answers to the Lisp-specific

TABLE VI

DATA COLLECTED FROM THE COMMENT SHEETS

Question (Answers on a scale of 0–5)	Manual Mean	Adaptive Mean	p-value
Rate Lisp as a programming language.	1.90	3.36	0.01
How often is the help screen consulted?	3.16	3.91	0.04
Rate COACH as a learning environment.	1.97	2.97	0.05
How helpful is help screen?	2.44	3.20	0.11
Is COACH better than a line-based environment?	3.31	3.49	0.70

Notes: This shows that the students using the adaptive version of COACH liked Lisp more than the other group, consulted the help screen more often, and rated COACH higher as a learning environment. Although the students using the adaptive COACH tended to find the help screen more helpful than the other group, the difference was not significant. Notice that both groups rated COACH as better than a standard line-based environment. In the above table, p-value is the probability that the means in two samples are the same. This data is shown graphically in Fig. 11.

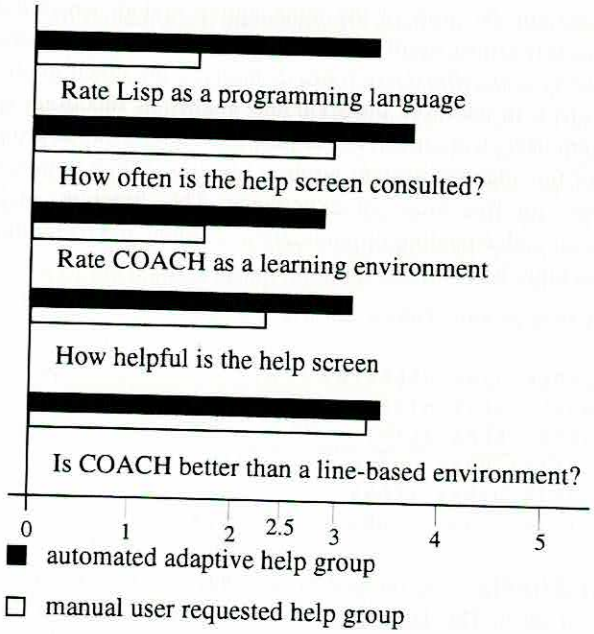

FIG. 11. Graphical depiction of comment data from Table VI.

questions showed that, except for the simplest cases, the students were unable to answer Lisp questions by guessing.

Saved Exercise Solutions. The COACH system internally stores information about each user. This internal representation is the student's adaptive user model (AUM). At the completion of users' work sessions, COACH creates two files. The first contains the user's work (Lisp code), and the second contains the user model (usage data for determining the level of the user's expertise for specific learnable units). Unfortunately, user model files were not kept for manual help students.

The students saved work and their saved user models showed that all students had finished the eight introductory exercises, i.e. those exercises which did not require defining functions. This was followed by a database project requiring students to write functions.

The comment sheets indicated that all the students had begun writing functions for the database project by the last session. Examining their work showed surprising differences in the percentage of the ten database project functions the students in the two groups had actually completed. The users of the adaptive system wrote an average of 2.5 functions, as compared to 0.5 for the users of the

nonadaptive system. No user of the nonadaptive system wrote more than two functions. This is the most significant result of the study: on the average, *the users of the adaptive system defined five times as many of the functions required in the data base project.* In addition, the style and quality of functions written by the adaptive system users were much better than that of the control group.

One user of the adaptive system wrote a function which demonstrated astonishing progress for five hours of experience. This function, included below, demonstrates an understanding of parameters, scoping and formatting, as well as boundary checking, DEFUN, and lists.

```
(DEFUN add-person (name phone lang)
  (COND
    ((MEMBER name USERS))
    ((EQUAL USERS nil)
      (SETQ USERS (LIST
        (LIST name phone lang))))
    (T (SETQ USERS (CONS
        (LIST name phone lang) USERS)))))
```

Comment Sheets. At the end of each day, the students were asked to fill out a comment sheet. The data from the comment sheets are the students' ratings of various aspects of the course. The values vary from zero to five, zero being the worst and five being the best. The statistical analysis was done using a two-tailed p-test, to determine the probability (p) that the mean rating of the users of the adaptive system was different than the mean of the control group. This was done using Welch's method (Brownlee, 1984) (see Table VI). The strongest result concerned users' ratings of Lisp as a programming language. The users of the adaptive system indicated that they had a higher regard for the Lisp language; the means of the answers for the two groups have a 0.01 probability of being the same as each other, giving a 99 % probability that the adaptive help group had a higher regard for Lisp than the control group ($p = 0.01$). The question, "How often do you look at the help screen while solving a problem?" showed that users of the adaptive system used the help screen more often ($p = 0.04$). The results from the question, "How helpful is the COACH window system, as compared to a line-based interpreted environment?" showed that both groups thought the COACH environment was an improvement over a standard interpreted environment. The mean rating for the question "How helpful is the help screen?" seemed to be higher for the users of the adaptive system; however, with the study sample size it did not prove to be significant ($p = 0.11$). This is probably because it is a hard question to answer. The question might have yielded better data if it had asked the students to compare the help screen to the tutorial or some other form of help.

Interview Tapes. Students were interviewed on audio tape during their

work sessions with standardized questions. Six randomly chosen individuals from each group participated in these interviews. The data collected from the interview tapes show differences in the ways the two groups utilized help while solving a problem (see Table VII and Fig. 12). All users reported the usefulness of menus to ask for help. The manual user-requested help group received help messages for syntax errors, the kind of help for which an interpreted Lisp environment is known. They reported that this computer presented help was not particularly useful. All users of the automated adaptive help reported finding it useful. While only one member of the manual group reported making use of the tutorial packet, all interviewed members of the adaptive group reported the tutorial useful. While

TABLE VII

STUDENTS USING DIFFERENT METHODS TO SOLVE PROBLEMS

Learning materials used	Manual	Adaptive
Asks COACH for Help	6	6
Uses COACH presented Help	0	6
Refers to tutorial	1	6
Uses trial and error	1	0

Notes: Of six students interviewed in the manual group and six students interviewed in the adaptive group, the adaptive help group found more of the support materials useful. This data is shown graphically in Fig. 12

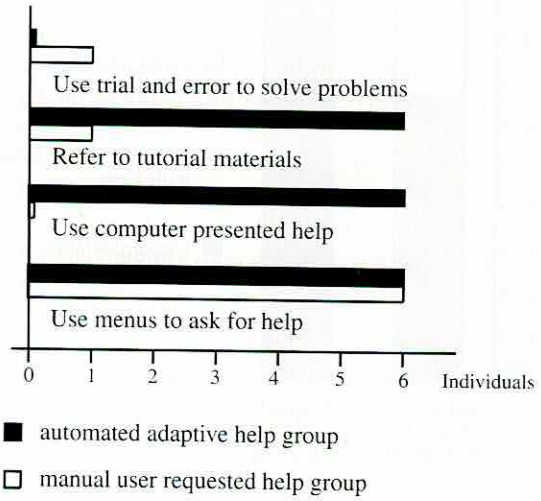

FIG. 12. Number of users reporting use of each method to solve problems. Data recorded from the six people interviewed from each of the two groups from Table VII.

one member of the manual group reported relying on trial and error to solve problems, none of the adaptive group reported using this technique.

In Fig. 12, it is notable that students in the adaptive system group used all the different types of help available to them, while students in the nonadaptive group did not.

Post-tests. A post-test was given to the students at the end of the course. The post-test asked the students how comfortable they felt with the Lisp language. The data collected from the post-tests show the difference in comfort levels between the two groups. Out of the six post-tests completed by users of the nonadaptive group, two students felt somewhat comfortable and four were uncomfortable. Of the nine post-tests completed by users of the adaptive system, three students were comfortable, five were somewhat comfortable, and one was uncomfortable (see Fig. 13).

8.2.3 Discussion

The data demonstrate differences in self-assessment and performance between users of an adaptive automated help system, as compared to users with manual help.

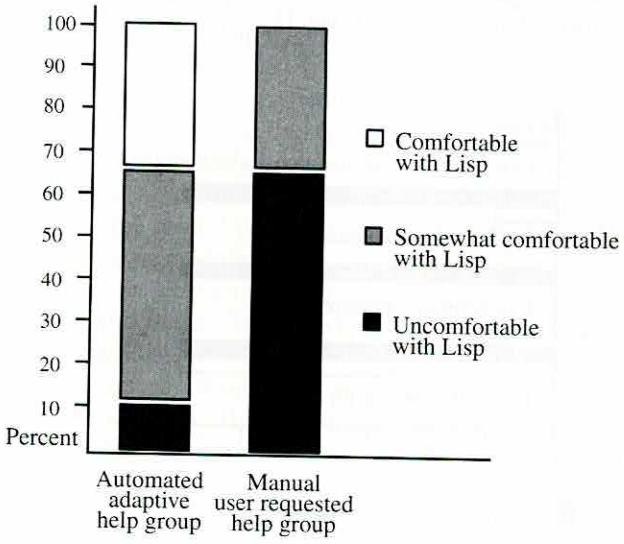

FIG. 13. Comfort levels reported by students at completion of course. Percentages are calculated for nine students from the adaptive automated help group and six students from the manual help group who returned the post-course questionnaire.

The terse nature of the tutorial purposely masked the quantity of information with which the users were familiarizing themselves. The amount of knowledge to which the students were exposed was close to what students might be expected to master in a full semester Lisp course.

Although the students were sometimes frustrated, they learned a lot of Lisp. The goal of requiring them to use the help system to solve their problems was achieved. Even though large performance differences were found between the groups, the nonadaptive group still performed extremely well. When compared to the amount of work a novice might accomplish in a typical learning environment, it is clear that the COACH environment was a significant aid. Even the nonadaptive users described their interface as an improvement over the usual tools that are available (Fig. 11).

The pressure of learning so much Lisp in such a short time without any human teacher help was a challenge. Although all students completed the study, one of the manual help group students required extensive persuasion to continue after the third day. A comparison of the full study to the pilot study showed that the pressure of the amount of material to be learned was decreased by the self-paced presentation.

8.3 Future Work

This study has shown that an adaptive automated help system can increase user performance. Many important questions remain unanswered (see Section 10).

To simplify the user study, the most experimental rule was eliminated from the COACH adaptive user model. A small number of adaptive rules were used to change the quantity and quality of help given to a user for each function, token, and concept. It is important to examine the various kinds of adaptation and knowledge bases in such an adaptive user interface. The value of each individual rule should be studied.

COACH also records user information concerning required and related knowledge. As described in Section 6.3.2 rules can use this information to describe alternative solutions or point to related *learnable units* when a user is struggling. For simplicity, the "encourage exploration" presentation rule was deactivated for the study. Additional rules could interject syllabi on which to tutor a user in such a situation (see Section 10). It would be interesting to study the value of such facilities, which may distract students from their task.

This study tested the effectiveness and importance of an adaptive system with Lisp-illiterate users. The ways in which COACH will be helpful to experts is probably quite different from the ways in which it will be helpful to novice programmers. Experts will benefit from COACH's "Level 3" examples and "Level 4" complete syntactic descriptions of functions. COACH leaves experts alone when they are working on something with which they are experienced. These

experienced Lisp programmers will benefit from the fact that COACH keeps track of context dependent situations, scope, and undefined variables, while exposing the user to the relationship between functions and concepts. Novices, on the other hand, find the changing adaptive help for functions and tokens quite useful for learning the syntax of simple functions and token types. While studies showing the different benefits for different users would be straightforward, they were beyond the scope of this experiment.

8.4 Conclusions

In this study, significant differences were found between students who used automated adaptive COACH help and students who had only manual COACH help.

While the responses to the comment sheet question concerning motivation during work sessions did not show a difference between the two groups, other indicators did. One might expect the group with less computer support to make greater use of the paper tutorial; however, the converse was true. Both groups had the same access to the paper tutorial and on-line help. While the group with manual COACH help only valued the user-requested help, the automated adaptive help group utilized all available materials, the Lisp tutorial and user-requested COACH help as well as automated coach help (see Fig. 12). Students from the adaptive group reported feeling more comfortable with Lisp and also completed many more of the exercises than the control group (see Fig. 13).

The automated adaptive help system succeeds in improving productivity and raising motivation to use available materials.

This section has described efforts to evaluate adaptive user help systems. The study raises many interesting questions for future research. The following section discusses these in more detail.

9. Development Status

The OS/2 WarpGuides product is based on a version of COACH rewritten in C++ and extended to support a graphical user interface. This graphical adaptive help system was written by Ron Barber with help from Bob Kelley, Steve Ihde and Julie Wright. The product uses innovative dialog masks and highlighting to draw attention to the graphical features being coached. Text balloons alongside but not obscuring the dialog box describe these features. In prototype versions, animation and sound have also been explored for augmenting the masks and text. These content types have been implemented with wav files for sound and a home grown animation language called GAS (Graphical Animation System).

9.1 Animated Help

We began analyzing the GUI to create graphical help that would best match the GUI interface using the elements of visual language formalism (Selker and Koved, 1988). Objects on the display define an alphabet of the graphical language. The operations that happen on them are the graphical syntax. For example, a folder is a terminal symbol, a double click is a parameter that is sent to it when the user double clicks. This syntax is interpreted by the system to cause the "open" semantic operation to occur. Describing operations in this way helped us to map these operations to animated presentations.

9.2 Slug Trails

Our first attempt to create example help to teach direct manipulation in a graphical user interface used a pictorial representation of a mouse, with buttons that changed color according to their state, as in (Goldberg and Robson, 1983). A procedure was shown in a static form by arcs between states of the mouse. This resembles the form of animation seen in comic strips. Temporal animation of the motion of the mouse and other objects made this seem more realistic. However, this did not leave a persistent record for the user to review of the important actions that had been shown in the animation. To make the animation easier to review and more concrete in the mind of the user, we developed "slug trails". The idea behind slug trails was to augment an animation showing a task with a persistent afterimage portraying the important actions. For dragging, the right button on our mouse would go down (shown depressed), the icon would be moved across the display leaving a trail of dots or other graphical debris behind it, then the button would go back up (shown normally). Remaining on the display were the important syntactic graphical actions. The icon at its first location, the mouse in its first important mode with the cursor on the icon, the dots showing where the icon moved and the final location with the final state of the mouse all remained on the display when the action was over. In this way, a movie also has a static presentation as a reminder of the important events that occurred in the movie. This technique was relatively effective when prototyped by creating a graphical program for OS/2 called ANIMATE and a language called GAS for writing such animations co-authored by John Haggis and Ron Barber. The animation was supplemented with text explaining "what" was being shown and "how" to do the graphical action. These elements work together to teach concepts. Breaking down the graphical procedure in this way made interpreting the actions concrete. These animations were played on a background image of a display screen. Our first problem with this, like other help systems, was that a tremendous amount of display area was devoted to this presentation. We sought ways to convey the same information with less dedicated space.

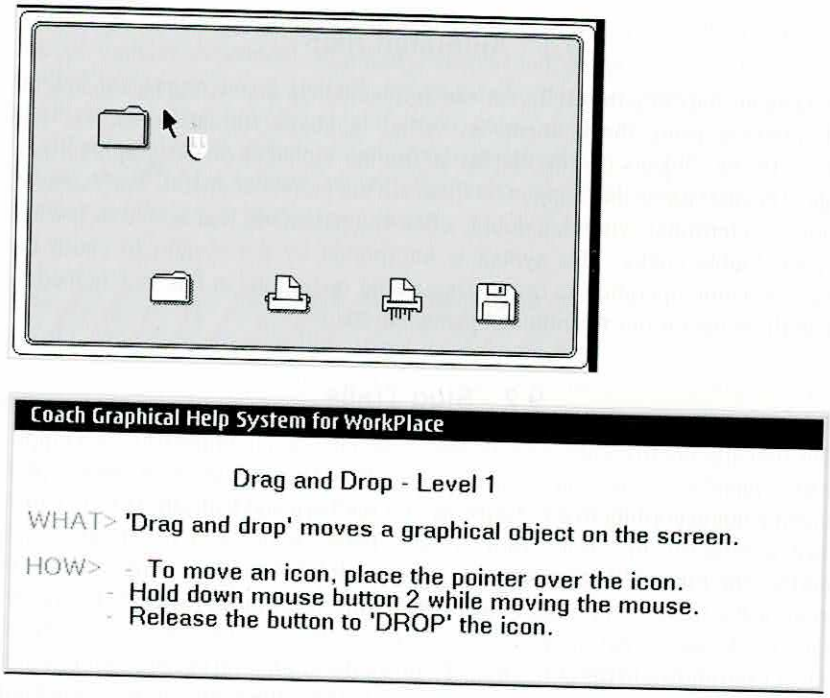

FIG. 14. The first frame of a "slug trails" animation, with the explanatory text which accompanies all frames.

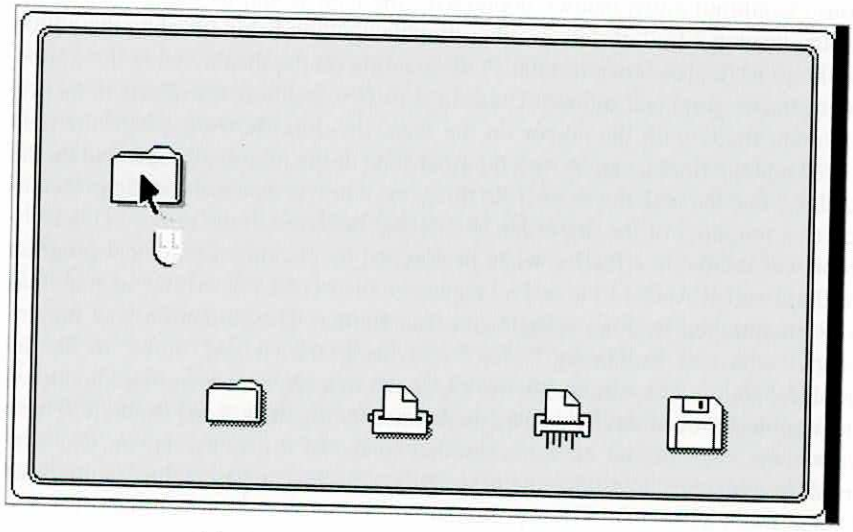

FIG. 15. The mouse moves to the top of the folder.

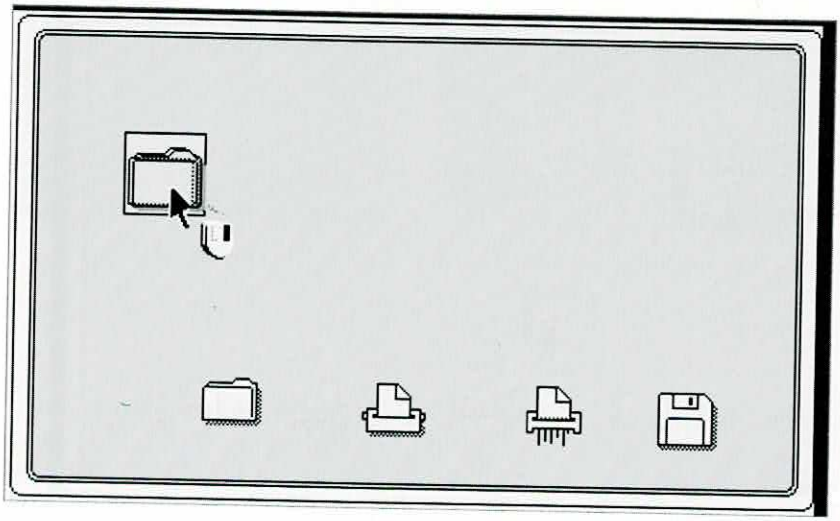

FIG. 16. The right mouse button is depressed, starting a drag.

9.3 Icon Dressing

After working with the slug trails for some time, we began exploring approaches that did not consume large amounts of screen real estate or draw focus away from their task. We wanted to develop lightweight presentation mechanisms that did not disturb the user's task. To address these problems, we defined a new idea called "icon dressing". Icon dressing is a form of graphical annotation which distinguishes an icon by outlining or otherwise embellishing it. This approach dresses it up to describe a likely operation that can be performed with it. We invented a vocabulary of small animations to prompt the user about appropriate actions involving the icon. These small animations focus the user's attention on the actual icon and what can be done with it. The icon dressing accomplished its goals; however, our prototype tended to require more interpretation by the user and was not flexible enough for general GUI assistance.

9.4 Cue Cards

We began designing a graphical presentation window with goals of associating information relative to an object temporally without covering up the interface with user controls. We realized this goal with the introduction of "Cue Cards". Distinctive coloration (typically yellow) and appearance (rounded corners, small, light, proportional fonts) like bubbles, together with minimal controls and quick

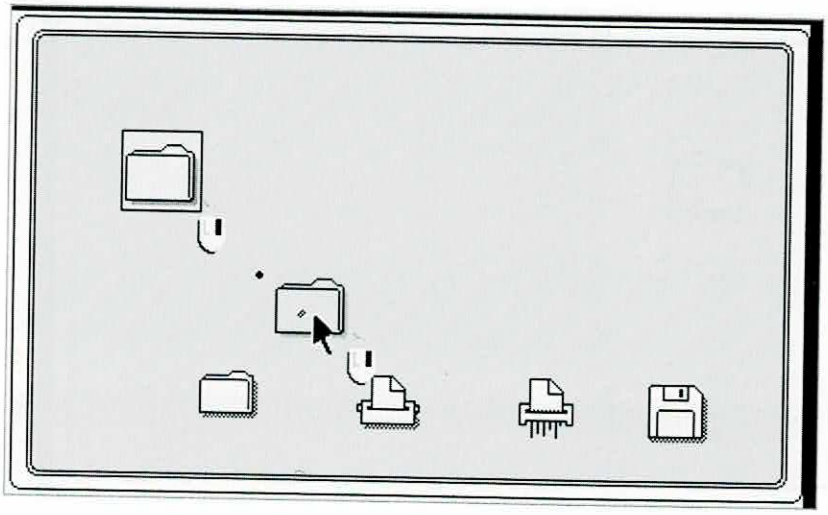

FIG. 17. In this frame, "slug trails" begin to form, showing a drag in progress.

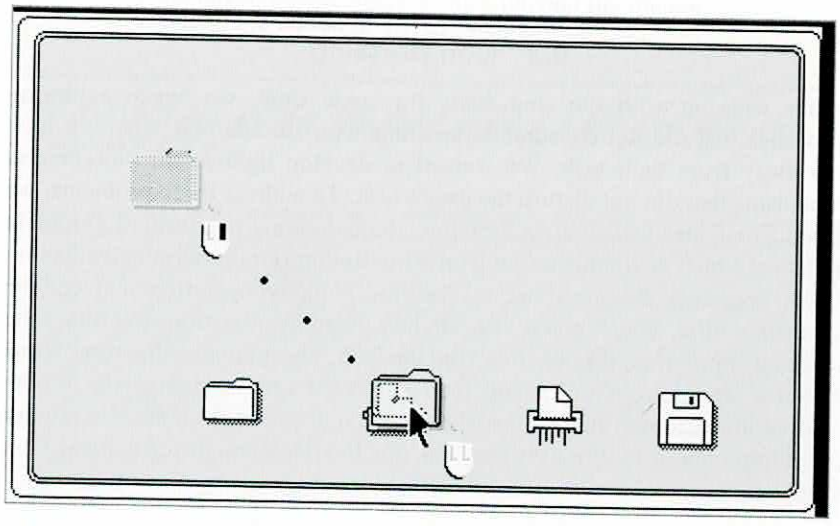

FIG. 18. In the last frame, the "slug trails" shows all the important steps of the action: click, move, and release.

response gave Cue Cards the feeling of supplementing or annotating the user interface without being so much a part of it. Cue Cards associate with something spatially and with color. We found that this feeling greatly contributed to the usability of our help system.

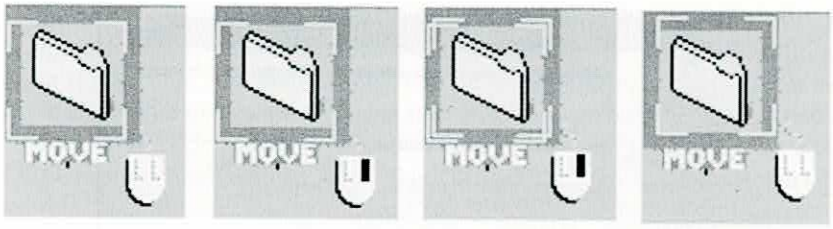

FIG. 19. An animated "icon dressing" shows the user how to move by dragging.

9.5 Guides and Masks

Dialog boxes are used to present some of the most complex and difficult system controls. One of the more useful applications of a GUI help system is to aid users with dialog boxes. Dialog boxes enhance the productivity of experienced users by presenting many options in a compact way, but their complexity can be frightening to the novice. Assistance agents such as Wizards (Mic, 1995) sidestep this problem, making it easier for new users to do more complex tasks such as installing printers and configuring the system, but they don't help the user learn to

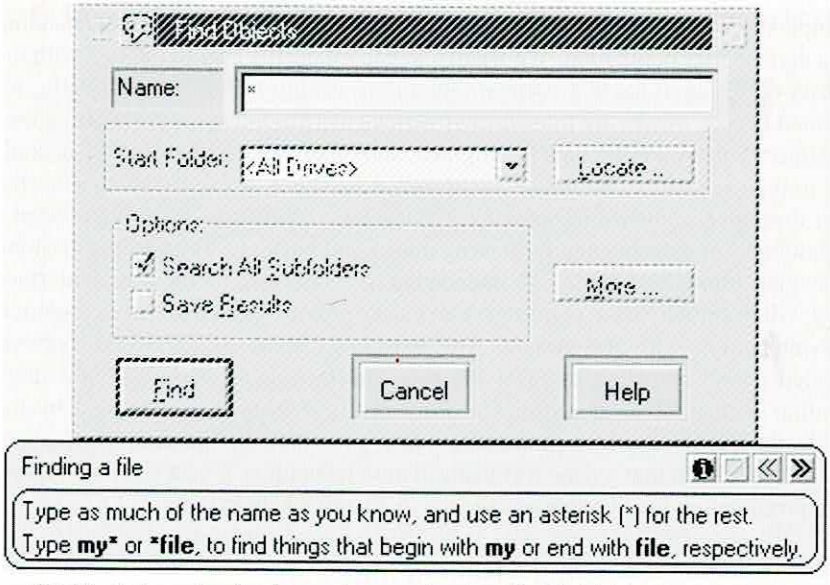

FIG. 20. At lower levels of expertise, attention is guided by highlighting important areas of the dialog and de-emphasizing the features less relevant to the learnable unit. No functionality is lost; the mask does not disable any features.

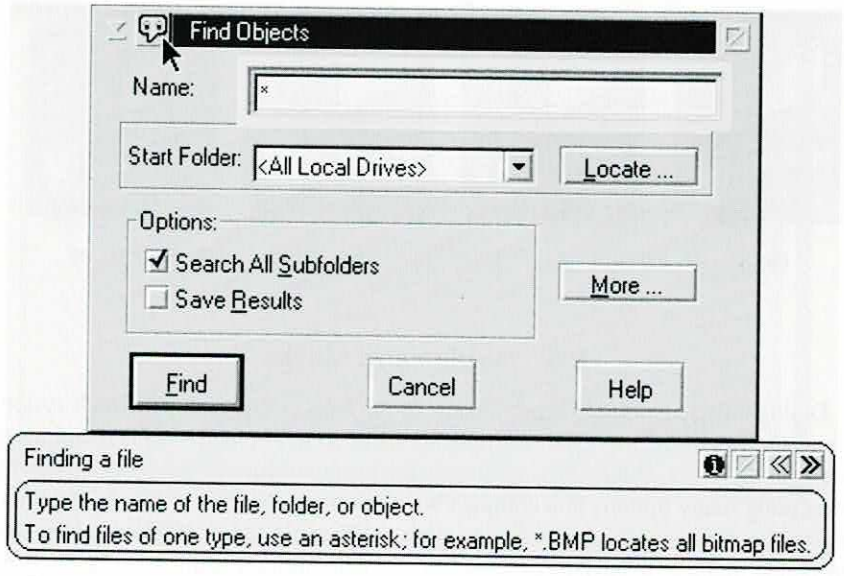

FIG. 21. As the user gains experience, the masks are discontinued, and more expert help text is provided.

use the native interface. Our idea was to make annotations that would vary the complexity of the dialog box interface according to the needs and understanding of a diverse user population. We wanted to encourage the user to interact with the subset of the interface that is important and to default the rest. Specifically, we defined masks. Masks are translucent overlays that shroud some parts of the user interface. Some parts become highlighted, and other parts are unaffected, according to their relevance. A "guide" comprises a sequence of masks which steps the user through a sequence of operations necessary to perform a task. We created a technology for defining and presenting masks and began testing various masking strategies. Masks and guides are intended to focus the user on the actual interface they will eventually have to master. The masks provide guidance without restricting interaction with the system. The masking can be progressively reduced (peeled away) revealing more of the user interface as the user becomes more familiar with it. Even if a user's activity diverges from that suggested by the guide, the COACH system continues to show a cue card and mask for the parts of the user interface that a person is using. These techniques give a flexible expressive presentation medium for our proactive adaptive help system.

9.6 Hailing Indicator

The proactive nature of COACH requires a way to communicate the availability of help information to the user. To accomplish this we added the "hailing

indicator", a small icon which may appear in the title bar of a dialog box. If a hailing indicator is shown, it has two distinct appearances: one notifies the user that a single guide is available for the current (unambiguous) task, the other notifies the user that guides are available for more than one possible task. When the COACH user model shows that the user has little recent experience with a task, COACH presents a suitable guide. Otherwise, just the hailing indicator will come up, reminding the user that help is available. Clicking on the hailing indicator for a single guide will bring up the guide. The multiple hailing indicator will bring up a guide menu, so the user can indicate which guide is relevant.

9.7 Sound

Finally, we are experimenting with the use of sound to supplement the other techniques. The use of sound can be particularly valuable to introduce the interface and the help mechanisms, creating the initial associations between kinds of help and their uses, and also between icon dressing animations and their meanings. The adaptive technology of COACH enables the sound annotation to become less verbose or disappear when it is no longer useful.

9.8 Summary

Widgets, graphical presentation techniques, and audio presentation techniques are a large and exciting field. Our work has emphasized presentation techniques

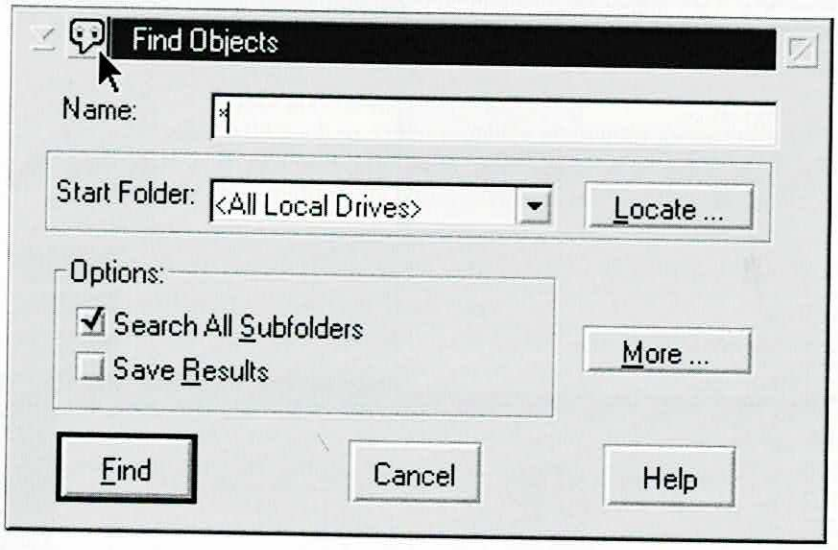

FIG. 22. Users can click COACH/2's hailing indicator (in the upper left corner) to activate the advisory agent.

indicator", a small icon which may appear in the title bar of a dialog box. If a hailing indicator is shown, it has two distinct appearances: one notifies the user that a single guide is available for the current (unambiguous) task, the other notifies the user that guides are available for more than one possible task. When the COACH user model shows that the user has little recent experience with a task, COACH presents a suitable guide. Otherwise, just the hailing indicator will come up, reminding the user that help is available. Clicking on the hailing indicator for a single guide will bring up the guide. The multiple hailing indicator will bring up a guide menu, so the user can indicate which guide is relevant.

9.7 Sound

Finally, we are experimenting with the use of sound to supplement the other techniques. The use of sound can be particularly valuable to introduce the interface and the help mechanisms, creating the initial associations between kinds of help and their uses, and also between icon dressing animations and their meanings. The adaptive technology of COACH enables the sound annotation to become less verbose or disappear when it is no longer useful.

9.8 Summary

Widgets, graphical presentation techniques, and audio presentation techniques are a large and exciting field. Our work has emphasized presentation techniques

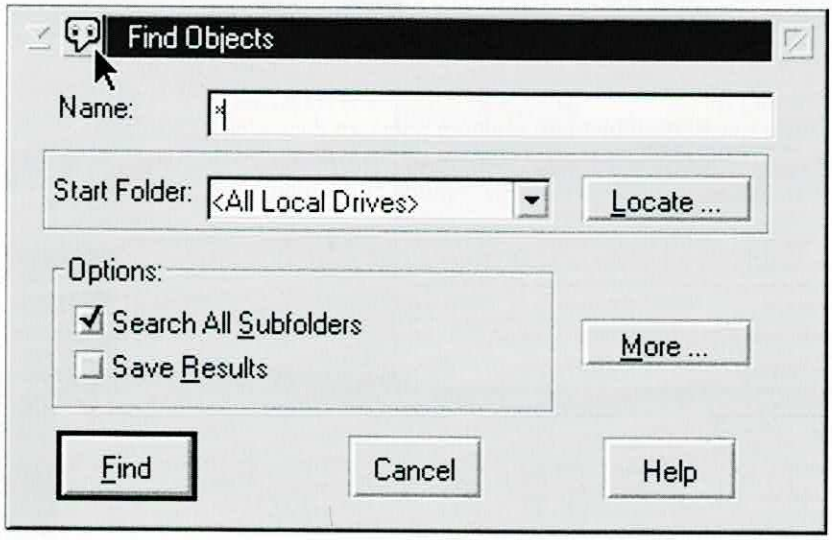

FIG. 22. Users can click COACH/2's hailing indicator (in the upper left corner) to activate the advisory agent.

for communicating temporal and associated information while not distracting a user from the tasks in a GUI. In our search for such techniques we have developed slug trail animations, icon dressing, cue cards, masks guides, and sound for presenting adaptive proactive help. Slug trails are a modality for graphically demonstrating an example of a procedure. Icon dressing accomplishes the same thing with much less visual real estate at the cost of design flexibility and versatility. Masks are a technique for simplifying the presentation of any user interface, focusing a user's attention on specific things. Cue cards are lightweight, versatile help presenters designed to be associated with but not cover up user interface function. Recognizing the strengths of each of these techniques has guided the choice of the COACH/2 help presentation mechanisms.

A version of COACH/2 ships with the OS/2 operating system starting with release Warp 4, under the name WarpGuide. A check-in procedure identifies the user and selects the appropriate AUM from a database of users. An authoring tool has been used for experiments with using COACH-based help technology with application programs. Figure 23 shows the authoring tool in use. Three windows are open: a tool bar at the top, a text-input window on the left, and a dialog box being annotated on the right. The dialog box appears as it would look to the user,

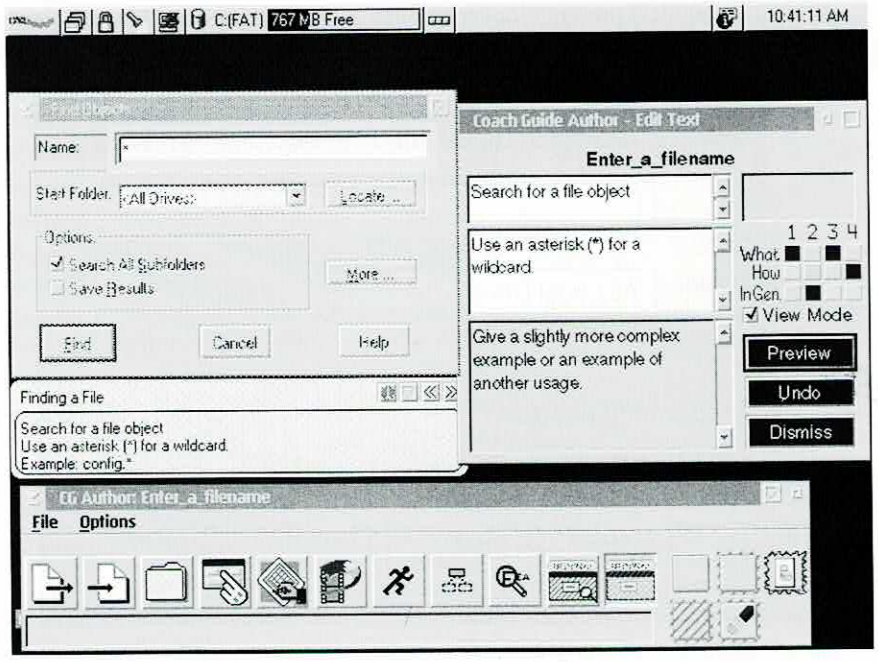

FIG. 23. COACH/2 WYSIWYG authoring tool.

with masking and highlighting to emphasize a region for entering a file name. Below the dialog box is the help text as it would be presented to the user.

COACH/2 monitors user events in the operating system's event queue, therefore COACH-based help can be added to any application that uses the operating system's API without modifying that application. However, supporting the AUM requires the application to provide a success/failure signal and an identifier for the step within a task where a problem was encountered. Without this support, COACH relies only on how a user "touched" a task step to update the user model.

10. Future Research Goals

COACH was originally developed as a research tool to explore teaching approaches, adaptive interfaces, and learning paradigms. The User System Ergonomics Research (USER) group at IBM Almaden Research Center is currently using COACH to explore several new ways of using and expanding the technology. The following is a short list of directions that are or could be followed using COACH as a research vehicle:

- Further experiments could establish the validity of particular *instructional techniques* in specific situations.
- Specific *adaptive mechanisms* should be more fully studied to establish their impact on the user.
- Currently COACH gives help advice addressing user problems it identifies. The use of COACH to actually perform the solutions to these problems as an *assistance* agent could save users from rote work, although the impact on learning should be studied.
- *Multi-media* help such as video graphics and audio are being tested with COACH/2. Experiments should be run to show if and where users derive more value from multimedia prompts and visual presentation than from text balloons.
- Tutorial curricula is being added to COACH to make it useful for teaching a syllabus.

As well as being a research platform, COACH is already intrinsically useful as a coaching interface. Development work with COACH/2 is being performed on popular operating systems (OS/2, Windows95). Authoring tools currently under development facilitate support for COACH on applications, which could be used to create a uniform help system for the entire environment experienced by the user.

10.1 Future Research

This section discusses future directions for COACH research in more detail.

(1) Evaluating *instructional techniques.*

Various basic instructional techniques can be embodied in a computer coaching system. In a useful exploration of the goals and techniques of teaching, Alan Collins and Albert L. Stevens (Collins and Stevens, 1983) put forward a list of ten such techniques demonstrated in computer teaching systems:

(a) selecting positive and negative exemplars;
(b) varying cases systematically;
(c) selecting counter-examples;
(d) generating hypothetical cases;
(e) forming hypotheses;
(f) testing hypotheses;
(g) considering alternative predictions;
(h) entrapping students;
(i) tracing consequences to a contradiction;
(j) questioning authority.

A fruitful area of research would be the formal evaluation of these instructional techniques. It remains to be proven which are most effective, which can be used together, and, most vital, which are appropriate for a particular situation. COACH would allow teaching techniques to be tested objectively. They would be embodied in coaching rules.

(2) How do *adaptive mechanisms* impact the user?

The adaptive strategies present in the COACH system were arrived at by informal experimentation. Guinea pig users worked with the system to test various adaptive and presentation strategies. A researcher can change rules to alter the system's coaching actions and strategies (see Section 7). Formal studies to determine which strategies are best could be set up to address issues in education, cognitive psychology, and cognitive science. Many questions could be easily tested. For example, is it better, when users are first exposed to a performance help level, to show them syntax and description, or would it be better to simply focus on an example? Presently the system waits to show related knowledge until the user has shown experience with the learnable unit. The current hypothesis is that too much information might overwhelm a beginner. A different hypothesis might state that the novice should instead be provided as much information as possible when just getting started. Do tasks that are done frequently get a greater benefit from an advisory style of help, while tasks which are rarely done get less value from the teaching aspect of the help system? Exploring such questions in more detail would give insight to

better understand student cognitive models. COACH is designed to explore such issues.

(3) The use of *agents* in an adaptive teaching interface.

Should the computer tell a user how to do something, or should the computer do it for them? An *advisory* agent that offers advice which the user is free to ignore is less obtrusive than an *assistant* style agent which implements the advice. In an assistance agent paradigm, when the computer knows something needs to be fixed by a user (e.g., inserting an open parenthesis), the computer takes control and types the solution. For example, in such a paradigm, when the computer identifies some way of simplifying what a user needs to do, it might create a macro so that the complex utterance can be described in a simplified way. The computer, then, has built a new instruction to come to the aid of the user. The computer is building a private, helpful set of tools for the user, a language the user and the computer both know. While the things the computer tries to do to the user's interaction with a program are what the user really needs and wants, and to the extent that the user can easily ask for advice or the computer can recognize the need for it, an assistance agent can be helpful.

The original COACH interface did not implement assistance agents. The hypothesis was that if users do not have to perform things themselves, they will not learn them. Another reason for not including assistance agents in the coaching paradigm was based on the hypothesis that the private interface that assistance agents provide could be difficult for a teacher or colleague to understand when the computer failed to be of help.

(4) Integrating *multi-media* help.

Hardware and demonstrations integrating video and computers are becoming popular. An early work by Steve Gano, "Movie Manual", integrated text, menus, and video to demonstrate automotive repairs, such as changing the oil (Gano, 1982). Audio can capture attention, animation can demonstrate graphical actions, 3-d graphics can make help distinct from the user interface. Visceral media can attract attention and aid memory of learning experiences without being confused with other graphical devices. Experiments should be done to show whether these other media improve learning and comprehension, and how they are best used to complement other teaching modalities.

(5) Integrating *tutorial curricula* with COACH.

COACH demonstrates an AUM-based teaching aid based on the assumption of a goal-directed user. While most of people's lives are spent trying to achieve their own goals, we all go through a period of schooling—a

time when others define our goals. The coaching interaction style supports a user's goals, leaving any "syllabus" or goal definition up to the user or a human teacher.

COACH/2 has guides as well as learnable units. A guide is composed of a single thread or path through a series of learnable units. Features can be defined to trigger advancement to the next learnable unit in the series, or the user can explicitly call for the next unit using the navigation buttons on the cue card. Multiple guides can share the same learnable units, e.g. the learnable units for naming a file would be part of guides for both opening a file and saving it under a new name. The author clicks on buttons to indicate the level of user knowledge appropriate for the learnable unit (four levels are supported) and the type of help information being provided (what, how, or general information).

The architecture could easily support more directed teaching materials, as demonstrated in systems like the Lisp Tutor (Reiser *et al.*, 1985). The network of relationships between learnable units in COACH could be made to work with a check list (an overall basis-set, see Section 6.3), to go through teaching materials in a sequence. Specific syllabus teaching materials with assignments and problems could be represented in domain knowledge frames (see Section 6.3.1). Adding a rule "what-to-teach-next" to the COACH rule set (see Section 6.4), could interface between the syllabus and the AUM to select appropriate teaching materials.

Such a system would have curricular goals, a syllabus, and the ability to evaluate a user relative to these goals. In addition to giving programmed lessons as other tutors do, such a system would be able to follow and help users in their programming, even when they were not doing exactly what is set forth in the curriculum.

Unlike a conventional syllabus, COACH allows the user to follow their own path through the material rather than enforcing a particular sequence. Adding tutoring techniques to COACH's presentation technology would allow for the kind of directed learning that syllabi create without requiring users to work through more of it than they need.

10.2 Future System Development

As well as being a research system, COACH has been used in real work. Below, two efforts are outlined which are making COACH more useful and available to users.

(1) *Integrating COACH into standard work environments.* COACH/2 is integrated into the OS/2 offering applications automatically; however, certain more ambitious integration projects could make the COACH architecture

more widely available:

(a) *User interface environment aids.* Rules could be added to COACH which provide already well-known agents. As demonstrated in Do What I Mean (DWIM) (Teitelman and Massinter, 1981), the system could correct spelling errors and correct variable naming or function naming. Assistance agents could also search for conflicts in other functions, similar function definitions, fix data type uses, etc. Assistance agents could also create solutions for equations and allow users to work with alternative representations, as demonstrated in the Mathematica (Wolfram, 1988) mathematical problem solving and visualization system.

(b) *Efficiency improvements.* Improved data structures help the COACH/2 system. The help text, rules, and user interface grammars of the Lisp implementation have been replaced in the COACH/2 implementation with a hashed object database. This greatly improves speed-of-access data structures for increased performance for giving help in standard GUI environments.

Improved algorithms also make the COACH/2 system more robust. The system caches text that is relevant to the particular user's AUM, leaving unneeded help information in files on disk. The COACH/2 system is approximately 0.5 megabytes of object code. Because of careful use of memory, it keeps its working set to this size as well. Currently, COACH/2 ships with about 1 megabyte of content.

In extremely complex programming exercises, identifying how to parse a code segment can be difficult. A more adaptive scoping parser could allow the COACH system to try multiple ways of parsing small pieces of users' work when users make changes.

(2) *Porting COACH for use in different environments.* Development work with COACH/2 is concentrating on implementations for the OS/2 and Windows95 operating systems, with the OS/2 version being shipped as WarpGuide with the OS/2 Warp 4 release.

Help support across different application environments raises interesting possibilities for transferring AUM data from one domain into another. Some of the AUM data that we have found easy to track and use in different parts of a single domain could be generally useful across domains:

(a) *User error rate.* The help system could have a lower threshold for questioning the accuracy of input by users known to be error prone. For example, a low score for typing accuracy earned in the text editor could cause the spreadsheet to be more sensitive to the entry of anomalous data, such as a four-digit number in a column of three-digit numbers.

(b) *Slow learners.* When a user is identified as a slow learner, the help system could concentrate on teaching the basic functions, while avoiding unsolicited advice on more obscure functions that may be confusing to the user. By developing this metric across different applications, appropriate help could be provided even on a user's first exposure to an application. The history of the user with other applications would substitute for a history with the new application.

(c) *Experimenters.* When the help system recognizes that a user quickly learns to use new features, it can be more forthcoming with hints and shortcuts. On the other hand, if a user is a non-experimenter, these helpful tips may be considered useless or annoying.

(d) *Preferred modality of interface.* Some users prefer keyboard input rather than using a pointing device. Some users make use of function keys, while others ignore them. Some users may have physical handicaps that make one modality difficult or impossible to use. A help system that can recognize a user's preferred style of interaction can suggest features that fit within those preferences and avoid mentioning features the user is unlikely to use.

(3) *Authoring tools.* In the original system an author wrote the structure of learnable units in a formal language. To facilitate support of COACH-based help for applications, an authoring tool has been developed for the OS/2 environment that allows a curriculum developer to describe graphical syntax with a WYSIWYG point-and-click interface.

A user selects authoring tasks from the authoring tool's GUI. It allows them to create new guides, to identify new learnable units, to give these guides and learnable units graphical looks, to edit and add content for them, and to identify the relationships between learnable units. An author selects the syntactic part of the GUI for which they want to create help and uses the guide author to create content while demonstrating the actual use of the thing to be learned. The authoring tool automatically creates the COACH syntactic model, allowing it to notice when to give help on the learnable unit. In this way the author defines a mask, features that show through the mask, highlighted features, and a cue card for each learnable unit.

A completely new implementation of the authoring tool is being developed in Java, to take advantage of the cross-platform portability of that language.

(4) *Web implementation.* COACH could be implemented for Web-based applications. The interaction between the user and the advisory agent is implemented as a client–server relationship in COACH. In our UNIX-teaching COACH, the help agent ran on a different computer from the one

running UNIX. A Web-based COACH could have the same structure, in which a COACH server could download applets to the user's machine to provide text, graphics, sound, and other forms of help information. Alternatively, COACH itself could run only on the client in applet space or as part of a proxy server.

A Web COACH could explain interfaces with complex syntax, such as the wild cards and logical operators used to request information from search engines. It could warn about unintended side effects, such as buttons which download software and change the operation of a user's machine. If there are plug-ins or upgrades required to implement certain features of a Web site, COACH could guide the user through installation of the software. COACH could simplify Web page design because information that isn't relevant to all users would be hidden from view unless and until an appropriate moment to present it occurred. COACH would streamline access to the information content on the Web, by selectively controlling the content seen by the user, as appropriate for the user's browsing experience with each site.

ACKNOWLEDGMENTS

This chapter would have never been completed without the tireless editing and dedication of Mark Thorson. This chapter is based on my PhD dissertation, a document that many people, especially my wife Ellen Shay and sister Diane Selker, worked hard to help me accomplish. Many people have contributed to COACH since the original dissertation; Bob Kelley, Steve Ihde, Julie Wright, and John Haggis worked with Ron Barber to create the COACH/2 implementation. As with any product, a large number of people who worked hard to support the productization of COACH. Without Les Wilson, Ashok Chandra, and Maria Villar, it would not have happened. Ron Barber's work stands alone. His architectural work, long nights of programming, and leadership made the product happen.

REFERENCES AND FURTHER READING

Alexander, S. M., and Jagannathan, V. (1986). Advisory system for control chart selection. *Computer And Industrial Engineering*, **10**(3).

Aronson, D., and Briggs, L. (1983). Contributions of Gagné and Briggs to a prescriptive model of instruction. In *Instructional-Design Theories and Models: An Overview of Their Current Status*, pp. 75–163. Lawrence Erlbaum Associates, New York.

Barr, A., and Feigenbaum, E. A. (eds.) (1984). *The Handbook of Artificial Intelligence*. William Kaufmann, Los Altos CA.

Bereiter, C., and Scardamalia, M. (1984). Learning and comprehension. In *Information Processing Demand of Text Composition*, pp. 407–421. Erlbaum, Hillsdale, NJ.

Bonar, J., and Soloway, E (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, **1**, 133–161.

Borenstein, N. S. (1985). The Design and Evaluation of On-line Help Systems. PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.

Brehm, S., and Brehm, J. W. (1981). *Psychological Reactance: A Theory of Freedom and Control*. Academic Press, New York.

Brownlee, K. A. (1984). *Statistical Theory and Method in Science and Engineering*. Krieger, Malabar, FL.

Burton, R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, **2**.

Burton, R. (1982). *Intelligent Tutoring Systems*, chapter 4. Academic Press, New York.

Burton, R., and Brown, J. S. (1982). *Intelligent Tutoring Systems*, chapter 2. Addison-Wesley, New York.

Campbell, R. L. (1989). Developmental levels and scenarios for smalltalk programming. Technical Report RC15305, IBM T. J. Watson Research Center, Yorktown Heights, NY, December.

Campbell, R. L. (1990). Online assistance: conceptual issues. Technical Report RC15407, IBM T. J. Watson Research Center, Yorktown Heights, NY, December.

Carbonell, J. G. (1970). An artificial intelligence approach to computer assisted instruction. *IEEE Transactions on Man–Machine Systems*, MMS-**11**(4).

Carbonell, J. G. (1979). Computer models of human personality traits. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh PA.

Carroll, J. M., and Aaronson, A. (1988). Learning by doing with simulated intelligent help. *CACM*, **31**(9).

Clancy, W. (1986). From guidon to neomycin and heracles in twenty short lessons: Orn final report 1979–1985. *The AI Magazine*, pp. 40–60, August.

Collins, A., and Stevens, A. L. (1983). Instructional-design theories and models: An overview of their current status. In *Cognitive Theory of Inquiry Teaching*, pp. 247–229. Lawrence Erlbaum Associates, New York.

Conklin, J. (1986). A survey of hypertext. Technical Report STD-356-86, MCC, Austin TX, October.

Corbett, A. T., and Anderson, J. R. (1989). Feedback timing and student control in the Lisp intelligent tutoring system. In *Proceedings of The International Conference on Artificial Intelligence*, pp. 64–72. IOS, Amsterdam.

Davis, R., and Shortliffe, E. (1977). Production rules as a representation for a knowledge-based consultation system. In *CHI '87 Proceedings*, Vol. 8, pp. 15–45.

Ellis, T. O., and Sibley, W. L. (1966). The grail project. *Spring Joint Computer Conference, Boston, MA*. Verbal and film presentation.

Erman, L. D., and Lesser, V. (1975). A multi-level organization for problem solving using many diverse, cooperating sources of knowledge. In *IJCAI*, Vol. 4, pp. 483–490.

Feldman, D. H. (ed.) (1980). *Beyond Universals In Child and Adult Development*. Ablex, Norwood NJ.

Fikes, R., and Keeler (1985). The role of frame based representations in reasoning. *CACM*, **28**(9), 904–920, September.

Fischer, G., Lemke, A., and Schwab, T. (1985) Knowledge-based help systems. In *CHI Proceedings*.

Gano, S. (1982). Movie manual. Technical report, MIT Media Lab, Cambridge MA.

Genesereth, M. (1982). Intelligent Tutoring Systems. In *The Genetic Graph*. Addison-Wesley, New York.

Gentner, D. R. (1986). A tutor based on active schemas. *Computational Intelligence*, **2**.

Glaser, R. (1985). Thoughts on expertise. Technical Report AD-A157 394, Learning Research and Development Center, Pittsburgh, PA, May.

Goldberg, A., and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, New York.

Grise, R. F., Jr. (1986). ANGEL: A pleasant user-interface for an interactive computing environment. Master's thesis, Cybernetic Systems Department, San Jose University, San Jose, CA.

Heidegger, M. (1977). *The Question Concerning Technology*. Harper & Row, New York.

Hewitt, C. (1972). Description and theoretical analysis (using schemata) of planner, a language for proving theorems and manipulating models in a robot. Technical Report TR-258, MIT A.I. Laboratory, Cambridge MA.

Hewitt, C (1985). The challenge of open systems. *Byte Magazine*, pp. 223–342, April.

Hopcroft, J. E., and Ullman, J. D. (1979). *Introduction to Automa Theory, Languages, and Computation*. Addison-Wesley, New York.

Houghton, R. C. (1984). On-line help systems: A conspectus. *CACM*, **27**(2).

Kernighan, B. W., and Pike, B. (eds.) (1984a). *The UNIX Programming Environment*, pp. 240–255. Prentice-Hall, New York.

Kernighan, B. W., and Pike, B. (eds.) (1984b). *The UNIX Programming Environment*. Prentice-Hall, New York.

Lawrence, K. (1984). Artificial intelligence in the man/machine interface. *Data Processing*, **1**, 231–236.

Lieberman, H. (1985). There's more to menu systems than meets the screen. In *Proceedings of the ACM/SIGGRAPH Conference*, Vol. 24.

Lieberman, H., and Hewitt, C. (1980). A session with tinker. Technical Report 577, MIT AI Laboratory, Cambridge MA, September.

Mackinlay, J. (1986). *Automatic Design of Graphical Presentations*. PhD thesis, Computer Science Department, Stanford University, Stanford CA.

Malone, T. W., and Lepper, M. R. (1987). Making learning fun: a taxonomy of intrinsic motivation for learning. Lawrence Erlbaum Associates, New York.

Manna, Z. (1972). *Mathematical Theory of Computation*, chapter 5–3. McGraw-Hill, NY.

Mastaglio, T. (1989). Tutors coaches and critics. Technical report, Computer Science Department, University of Colorodo, Boulder CO.

Mays, E., Apte, C., Griesmer, J., and Kastner, J. (1988). Experience with K-Rep: An object-centered knowledge representation. In *The Fourth Conference on Artificial Intelligence Applications, Proceedings*. IEEE Computer Society Press.

Merrill, M. D. (1983). Component display Theory. In *Instructional-design Theories and Models: An Overview of Their Current Status*, pp. 279–334. Lawrence Erlbaum Associates, New York.

Microsoft Corp. (1995). *Windows 95 User's Guide*.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1983). *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company, Palo Alto, CA.

Minsky, M. (1976). Frames. Technical report, AI Laboratory, MIT, Cambridge MA.

Moon, D. (1987). *User's Guide To Symbolics Computers*.

Morris, N. M., and Rouse, W. B. (1986). Adaptive aiding for human–computer control: Experimental studies of. Technical Report AAMRL-TR-86-005, Armstrong Medical Research Laboratory, Wright-Patterson Air Force Base, OH.

Myers, B. A. (1986). Visual programming, programming by example, and program visualization: A taxonomy. In *CHI '86 Proceedings*, pp. 59–66.

Pirolli, P. (1986). A cognitive model and computer tutor for programming recursion. *Human–Computer Interaction*, **2**.

Rath, G. J., Anderson, N. S., and Brainerd, R. C. (1959). The IBM Research Center Teaching Machine Project. In *Automatic Teaching: The State of the Art*. John Wiley and Sons.

Reigeluth, C. M. (ed.) (1983). *Instructional-Design Theories and Models: An Overview of Their Current Status*. Lawrence Erlbaum Associates, New York.

Reiser, B. J., Anderson, J. R., and Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for lisp programming. *IJCAI*, **1**.

Reisner, P. (1986). Human computer interaction: What is it and what research is needed? Technical Report RJ5308, IBM Almaden Research Center, Almaden CA.

Revesman, M. E. (1983). Validation and Application of A Model of Human Decision Making For. PhD thesis, Industrial Engineering and Operations Research, Virginia Polytechnic, VA.

Rich, E. (1983). Users are individuals: Individualizing user models. *Int. Man–Machine Studies*, **18**, 199–214.

Rissland, E. (1978). Understanding mathematics. *Cognitive Science*, **2**, 361–383.

Schofield, J., Evans-Rhodes, D., and Huber, B. (1990). Artificial intelligence in the classroom: The impact of a computer-based tutor on teachers and students. *Social Science Computer Review*.

Selfridge, O. (1985). Personal communication.

Selker, T. (1989). Cognitive adaptive computer help (coach). In *Proceedings of The International Conference on Artificial Intelligence*, pp. 25–34. IOS, Amsterdam.

Selker, T. (1991). Cognitive adaptive computer help. Technical Report, videotape.

Selker, T., and Koved, L. (1988). Elements of visual language. *IEEE Workshop On Visual Languages*, October.

Sleeman, D., and Brown, J. S. (eds.) (1982). *Intelligent Tutoring Systems*. Academic Press, New York.

Snelbecker, G. E. (1983). Is Instructional Theory Alive and Well? *Instructional-design Theories and Models: An Overview of Their Current Status*, pp. 437–472. Lawrence Erlbaum Associates, New York.

Suppes, P. (1967). Some theoretical models for mathematics learning. *Journal of Research and Development in Education*, pp. 4–22.

Sussman, G., Winograd, T., and Charniak, E. (1970). Micro-planner reference manual. Technical Report AI Memo 203, AI Laboratory, MIT, Cambridge MA.

Teitelman, W., and Massinter, L. (1981). The interlisp programming environment. *Computer*, **14**(4), 25–34.

Vertelney, L., Arent, M., and Lieberman, H. (1991). Two disciplines in search of an interface: reflections on a design problem. In *The Art of Human–computer Interface Design*, Addison-Wesley, New York.

Waters, R. C. (1982). The programmer's apprentice: Knowledge based program editing. *IEEE Transactions on Software Engineering*, SE-**8**(1), 1–12.

Weiss, L. (1987). Conceptual model of an intelligent help system. Technical Report DDC/LW-15, ESPREE, May.

Whiteside, J., and Wixon, D. (1986). Improving human–computer interaction: A quest for cognitive science. Technical report, Digital Equipment Corporation, Maynard MA.

Winograd, T., and Flores, F. (1986). *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, Norwood NJ.

Winston, P. H. (1977). *Artificial Intelligence*. Addison-Wesley, New York.

Wolfram, S. (1988). *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, New York.

Zellermayer, M., Salomon, G., Globerson, T., and Givon, H. (1991). Enhancing writing-related metacognitions through a computerized writing partner. *American Education Research Journal*, pp. 373–391.

Zissos, A. Y., and Witten, I. H. (1985). User modeling for a computer coach: A case study. *Int. J. Man–Machine Studies*, **23**.