Cognitive Adaptive Computer Help (COACH): A Case Study

EDWIN J. (TED) SELKER

IBM Almaden Research Center
San Jose, California

Abstract

User interfaces can be difficult to master. Typically, when a user has a problem understanding the interface, computers respond with generic, difficult-to-interpret feedback. This is a case study of COgnitive Adaptive Computer Help (COACH), an example of a style of intelligent agent which gives more effective responses when problems occur in the interface between people and computers. It implements a *cognitive* interface which attempts to recognize the needs of a user and responds proactively as the user is typing. It records and analyzes user actions to *adapt computer* responses to the individual, offering useful *help* information even before the user requests it.

The approach uses dynamic models of both the user and the domain of knowledge the user is learning. These models teach and guide a user. COACH was first used in a study which validated real-time learning and reasoning in computer interface can improve users' productivity and comfort with an interface.

COACH was designed to facilitate development and study of adaptive help systems. The help given the user, the domain in which the user is being coached, even the way the system adapts to the user, are represented in frames and controlled by rules which can be changed.

COACH was first tested in teaching the problem domain of writing Lisp programs. To demonstrate the generality of COACH for teaching arbitrary problem domains, help systems were then created for the UNIX command language and GML text formatting language. In the original text-based COACH implementation, a new problem domain can be supported simply by defining the syntax of the domain and writing the help text.

The ideas have been applied to a modern graphical user interface (GUI) in a product version created for the OS/2 and Windows95 GUIs. This COACH/2 system uses new teaching techniques to graphically demonstrate GUI syntax and procedures. A see-through technique called masking draws users' attention to GUI objects. An animation technique called slug trails walks the user through graphical procedures.

COACH/2 also includes a WYSIWYG authoring approach and tool to extend the notion of the system being a shell for creating adaptive help systems. This tool automatically picks up syntax from the graphical interaction with the author, so only the help text itself needs to be written. The architecture is available as WarpGuides in the OS/2 Warp 4 release. WarpGuides show a user how to perform graphical actions in a graphical user interface.

ADVANCES IN COMPUTERS, VOL. 47
ISBN 0-12-012147-6

Current development work with COACH/2 is exploring 3-d animation, mixing adaptive computer help with adaptive tutoring.

1. Introduction

COgnitive Adaptive Help (COACH) is a user help system that monitors the interaction between the user and the computer to create personalized user help. Imagine learning a new operating system of programming language. COACH watches the user's actions to build an Adaptive User Model (AUM) that selects appropriate help advice. Just a football coach will stand on the sidelines and encourage, cajole, or reprimand, COACH is an advisory system that does not interfere with the user's actions but comments opportunistically to help the user along.

COACH chooses descriptions, examples, syntactic definitions, etc. as appropriate for user-demonstrated experience and proficiency. A description that advertises a command or function is helpful for getting started, but might become ignored if it is presented too often. An example showing how to perform a procedure is often valuable until the procedure is mastered, after which it is no longer useful and may even become annoying. A syntactic definition describing the generalization of the procedure becomes valuable when the procedure is close to being mastered.

Computer users find themselves needing classes, tutoring, help, and reference materials in order to be able to accomplish even the simplest of tasks with a computer. Terry Winograd and Fernando Flores's book (Winograd and Flores, 1986) discusses "breakdown" of readiness-to-hand in terms with which we are all familiar—a computer becoming the focus of attention because the user does not know how to proceed. The need to seek aid frustrates the user and prolongs the process of becoming proficient. Attempts to computerize teaching aids have created an active research field. The impact and acceptance of computers in teaching roles, however, continues to be elusive.

Creating computer interactions so natural that they require no outside learning (the "walk up and use" ideal) would allow all user effort to be focused on the primary task. The idea of a tool as an object that allows workers to concentrate on their task, rather than on the tool, was at the heart of Martin Heidegger's concept of "readiness-to-hand" (Heidegger, 1977; Winograd and Flores, 1986). COACH contributes toward that goal by giving more effective assistance to users while they try to focus on their work. Winograd (Winograd and Flores, 1986) describes the process of managing the "breakdown conversations" as the way to progress. When the productive conversation a user is having with the computer to accomplish a task breaks down, it sends the user into a new conversation to break through or go around the roadblock. COACH's goal is to manage these conversations when the computer's "unreadiness-to-hand" is slowing progress.

Unlike teacher-oriented learning paradigms, in which the teacher is driving the student, the design of COACH uses a style of teaching which is driven by the student. COACH facilitates a user's own objectives in a work session.

COACH is an example of an *advisory* agent, as contrasted with the *assistance* agent used in other help system and teaching implementations. An advisory agent shows users what they can learn. In contrast, an assistance agent does the task for the user. This advisory agent uses an *implicit* dialog with the user, working in the background, tracking user actions and looking for opportunities to present relevant, but unsolicited, advice. The user is free to ignore the advisory agent, as opposed to many implementations of agents that engage in an *explicit* dialog with the user.

The following are the research objectives addressed by COACH:

- To study the differences between an automated adaptive help system and a standard passive help system.
- To explore automated teaching technology that shifts the teaching paradigm away from a pre-structured format to concentrate instead on users' individual needs. COACH is an example of a teaching paradigm that moves toward an apprenticeship, or learn-while-doing, approach.
- To demonstrate the feasibility of an AUM-based advisory agent to guide selection of help advice without introducing unacceptable delays in system response. COACH is a demonstration of help system that adapts to a user while running concurrently with the software for which help is being provided. Earlier research proposed that an interactive adaptive teaching system was not feasible (Zissos and Witten, 1985). The COACH system demonstrated that an interactive adaptive teaching system is indeed feasible.
- To create a tool for enabling research in adaptive learning paradigms. The COACH architecture allows researchers to describe and test ideas about adaptation in learning and to develop adaptive teaching approaches. COACH allows researchers to build working adaptive help systems.

2. The COACH Scenario

Users need help while working on solutions to problems in a curriculum or attempting to do productive work using a computer. COACH aids the user in the mechanics of using the computer.

The computer creates an *adaptive user model* (AUM) of a user's experience and level of expertise. Machine-learning and reasoning techniques adapt the help provided to the needs of the particular user. Such help is said to be proactive when the computer anticipates the needs of the user and presents help before it is requested. Both the user and the computer can initiate help, in a *mixed initiative* interaction.

This learning paradigm is introduced by these three hypothetical users, working at different levels of experience and proficiency, learning the Lisp interpreted language.

2.1 A Novice Lisp Programmer

Freshman Bill is taking his first programming class. He has attended two classroom sessions and is sitting down for the first time in front of the computer. His assignment requires him to use Lisp s-expressions to make arithmetic calculations.

The computer screen with which he works is segmented into four panes: a user input pane to type and edit work, two help presentation panes (one general and one more specific) and an output pane (Fig. 1). To help Bill get started using the computer to do his work, the system encourages him to type an open parenthesis to begin an s-expression or to type a defined word (see Fig. 2). Examples show him this. He types (. The help changes, telling him that he must type a function name, and gives an example. "function", "s-expression" and "defined word" are concepts in the system's domain knowledge network. Bill remembers these words but does not quite remember what they mean. An example of the use of a function is displayed to get him started. Bill types **ADD**. The help window tells him that no known function starts with `AD`, and suggests that he press the "rubout" key. He could press the mouse button to browse available functions, but `PLUS`, not `ADD`, is the function he now remembers from his class and he types it (see Fig. 3). As

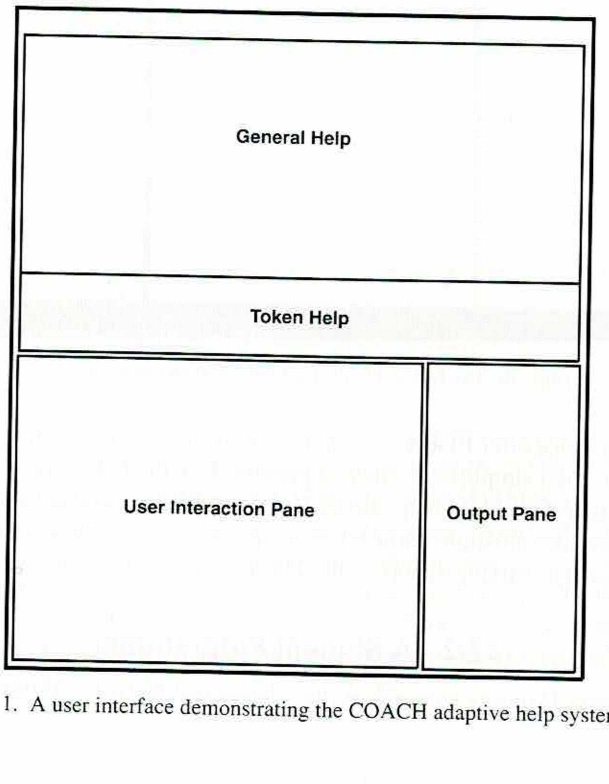
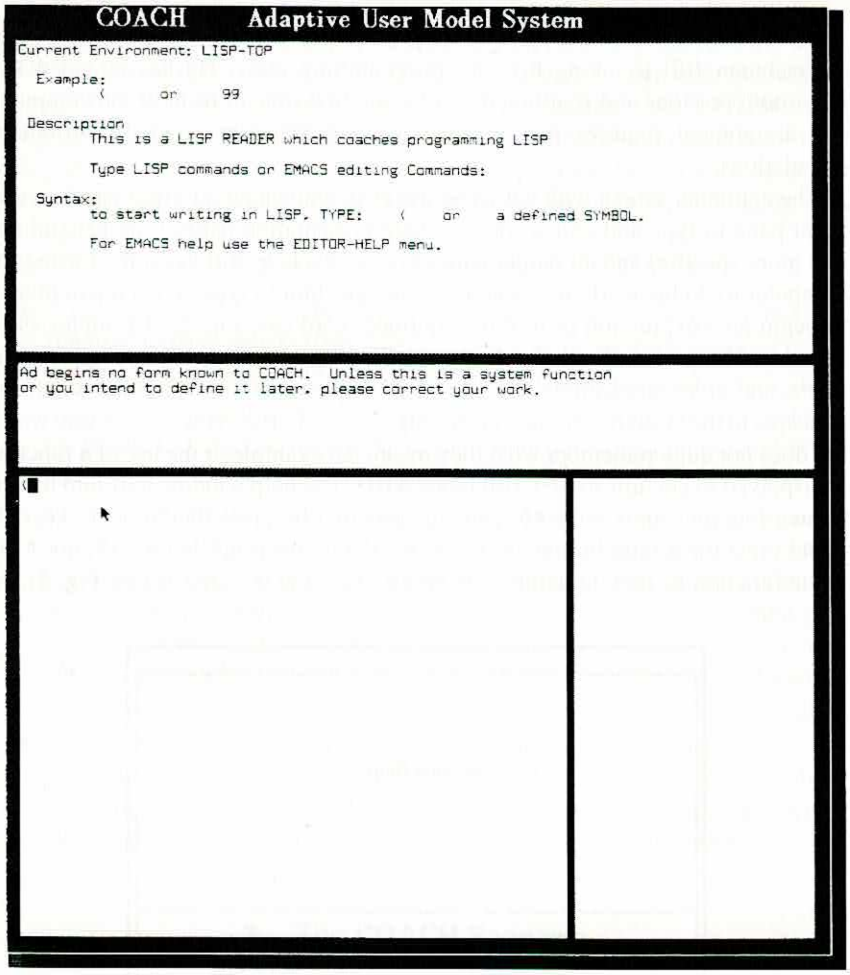


FIG. 1. A user interface demonstrating the COACH adaptive help system.



```
COACH      Adaptive User Model System
Current Environment: LISP-TOP
  Example:
          (        or      99
 Description
        This is a LISP READER which coaches programming LISP

        Type LISP commands or EMACS editing Commands:

  Syntax:
        to start writing in LISP. TYPE:     (     or     a defined SYMBOL.

        For EMACS help use the EDITOR-HELP menu.

Ad begins no form known to COACH.  Unless this is a system function
or you intend to define it later, please correct your work.

(
```

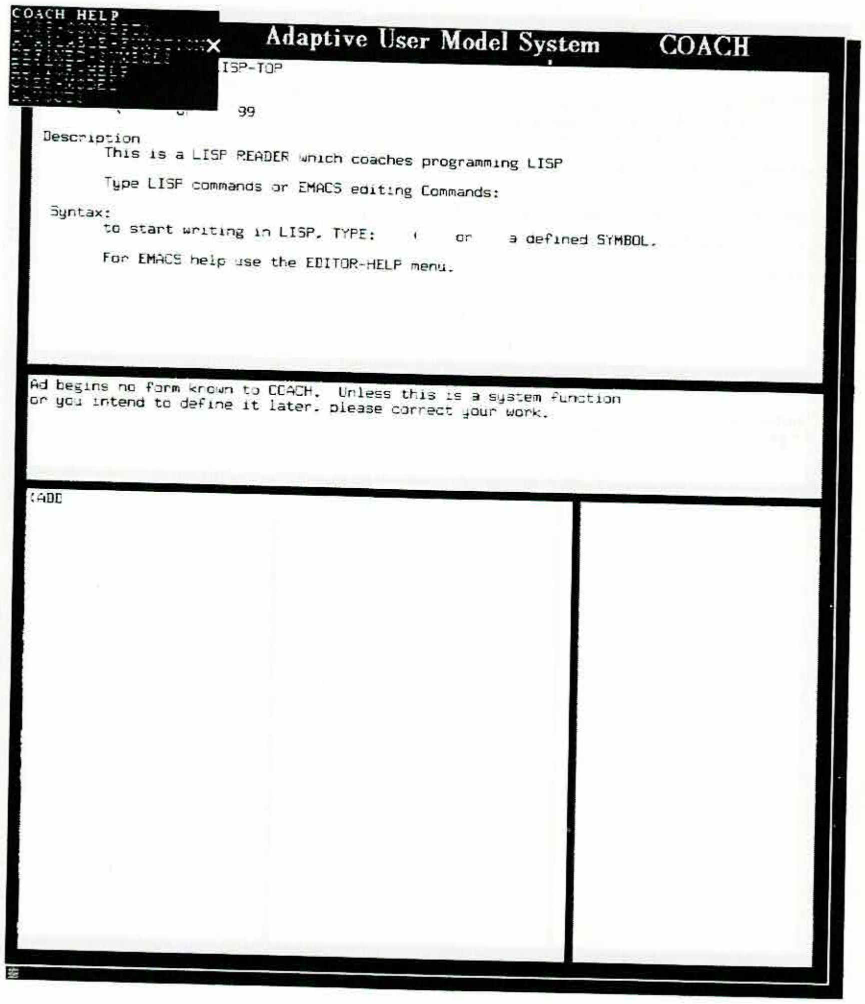
FIG. 2. The COACH interface after one character is typed.

Bill types a space after **PLUS**, an example of using `PLUS`, together with a simple description and a simplified syntax, is presented on the help pane.

The context-dependent help allows Bill to avoid the usual startup stalemate in which a user does not quite know what to type to get started. Novice programming problems such as mixing syntax with ideas have also been averted.

2.2 A Student Programmer

Sophomore Harry is trying to write a program. He types **DEFUN**. The help



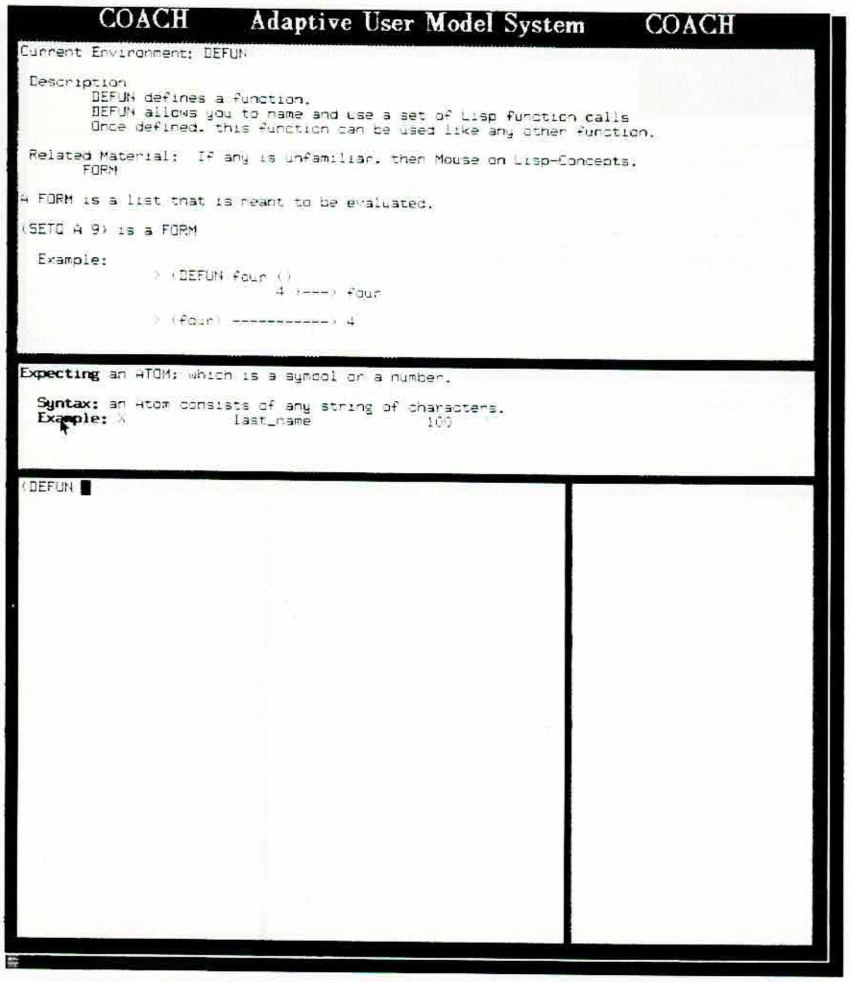
```
ISP-TOP

99
Description
      This is a LISP READER which coaches programming LISP

      Type LISP commands or EMACS editing Commands:
 Syntax:
      to start writing in LISP. TYPE:    (    or    a defined SYMBOL.

      For EMACS help use the EDITOR-HELP menu.
```

```
Ad begins no form known to COACH.  Unless this is a system function
or you intend to define it later. please correct your work.
```

```
(ADD
```

FIG. 3. The COACH interface during a simple error situation.

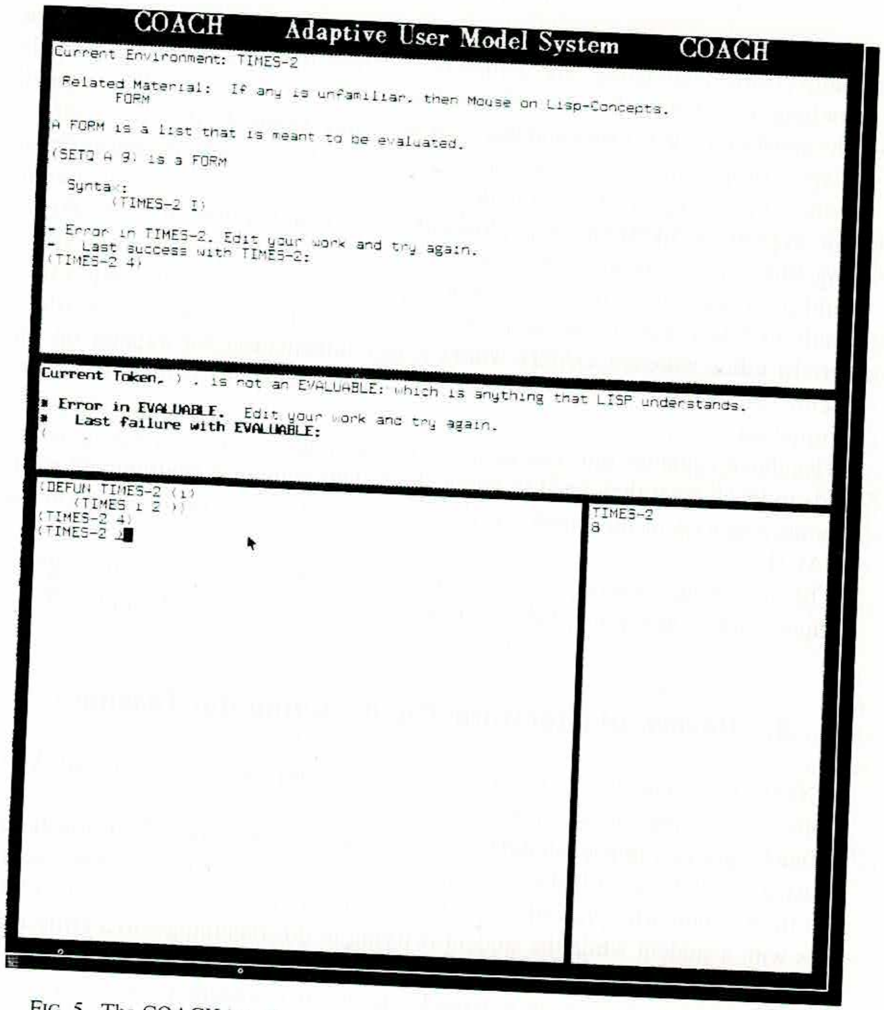
pane reminds him that he must name the function being defined and then give it an argument list. The system gives him an abbreviated syntax, omitting the difficult argument types (optional arguments, keyword arguments, etc.). He types **TIMES-2 (I) (PLUS**. The system shifts its focus to helping him with the `PLUS` function. An example of a use of `PLUS` he has already made is displayed. He realizes that he did not really mean to add numbers. He back-spaces and types **TIMES I 2)**. The system changes its focus of help to `TIMES` as Harry is typing it, and back to `DEFUN` when he is done with `TIMES`.



```
COACH          Adaptive User Model System          COACH
Current Environment: DEFUN

 Description
        DEFUN defines a function.
        DEFUN allows you to name and use a set of Lisp function calls
        Once defined. this function can be used like any other function.

 Related Material:  If any is unfamiliar. then Mouse on Lisp-Concepts.
       FORM

A FORM is a list that is meant to be evaluated.

(SETQ A 9) is a FORM

  Example:
                > (DEFUN four ()
                          4 )---) four

                > (four) -----------) 4

Expecting an ATOM; which is a symbol or a number.

  Syntax: an Atom consists of any string of characters.
  Example: X              last_name              100

(DEFUN █
```

FIG. 4. The COACH interface supporting learning about a specific Form and the idea of form.

Intermediate programmers like Harry often have problems keeping track of the context and appropriateness of program pieces. COACH works to keep this type of programmer oriented by providing context-sensitive help and user-examples. An instance of user-example help is shown in Fig. 5; the last correct use of `TIMES-2 (TIMES-2 4)` was presented when the user forgot to include an argument in the function call.



```
COACH      Adaptive User Model System      COACH
Current Environment: TIMES-2

 Related Material:  If any is unfamiliar, then Mouse on Lisp-Concepts.
      FORM

A FORM is a list that is meant to be evaluated.

(SETQ A 9) is a FORM

  Syntax:
      (TIMES-2 I)

- Error in TIMES-2. Edit your work and try again.
-    Last success with TIMES-2:
(TIMES-2 4)
```

```
Current Token, ) . is not an EVALUABLE: which is anything that LISP understands.
* Error in EVALUABLE.  Edit your work and try again.
*    Last failure with EVALUABLE:
(
```

```
(DEFUN TIMES-2 (i)
   (TIMES i 2 ))
(TIMES-2 4)
(TIMES-2 4
```

```
TIMES-2
8
```

FIG. 5. The COACH interface using automatically accumulated knowledge help for a user-defined form.

2.3 An Expert Programmer

Expert programmer Connie is working on an internal part of the `Update-Rule` computer program. A model of her expertise allows the system to know that it need present very little help. When she types (**SETF**, the system shows the very complex argument list syntax for the `SETF` function. If she uses a function for which no help has been written, the system reaches into that function's definition to present an argument list for it. If she makes an error (e.g., wrong argument type), the system changes its view of her expertise slowly at first. If she keeps making errors, it will change its opinion of her more quickly and begin to provide more help; it will show her examples and remind her of things which are related to the constructs she is using and the language concepts involved.

Expert programmers must be aware of anomalous, as well as simple, relationships between parts of a computer language. Even if they do not memorize them, experts are likely to use sophisticated syntactic features. If Connie were using function or variable names which she had not yet defined, the system would put these names on a list of undefined functions. A menu would allow Connie to select from those names to aid her in remembering to define them later. In other words, COACH would select information for experts on the obscure, anomalous parts of Lisp without bothering them with introductory information.

The above vignettes illustrate an adaptive user interface model tracking users' needs to teach what they need to know about their computer environment while they are engaged in their own work. A videotape (Selker, 1991) demonstrates COACH.

The following overview of relevant work describes the current state of computer aided instruction (CAI) and inspirations for the COACH approach.

3. Review of Literature: On-line Computer Teaching

COACH is a vehicle for research in human–computer interaction and AI as applied to teaching and learning.

Teaching styles impact students' roles in their learning tasks. Tom Mastaglio (Mastaglio, 1989) described a continuum of interaction styles for computer teaching: from a tutor who prescribes what a student should do—to a coach who kibitzes with a student while the student is trying to do something—to a critic who reviews work after it is completed. In these three teaching styles the point at which the system intervenes is varied relative to a student's phases of work: design, construction or evaluation.

The use of computer systems for teaching has been called computer aided instruction (CAI), intelligent computer aided instruction (ICAI), artificial intelligent computer aided instruction (AICAI), or intelligent tutoring systems (ITS). These names have been created by their proponents to reflect the technological and research progress through the years. A common component of all such work is a prescribed educational goal to which students are guided with subgoals and tasks. The curriculum can take the form of text with comprehension tests or problem sets or educational games. The term CAI will be used in this paper to refer to all such systems.

Research in CAI has included experiments using artificial intelligence (AI) representation, reasoning, and machine-learning techniques to direct a tutoring session (Reiser *et al.*, 1985; Sleeman and Brown, 1982). This section outlines progress in CAI, highlighting the roles AI has played. CAI teaching styles can be broken into the following classes:

- *Tutoring* systems which include a syllabus or courseware schedule of what a user should know when, and how to teach it. Such systems use generic models of what all students need to learn.
- *Help* systems which answer questions a user asks. These have no syllabus or model of what a student needs to learn and only interact with a user when the user requests assistance.
- *Coaching* systems which remark on user problems and successes as they occur. Like traditional help systems, coaching systems help users while they are working on problems. Unlike help systems, coaching systems offer unsolicited advice. A coaching system might also have mini-tutorials useful for specific situations in which users find themselves.
- *Critic* systems suggest or perform changes to completed user solutions. The original Lisp Critic (Fischer *et al.*, 1985) worked in this way, giving criticism and making improvements to work students brought to it. A critic allows users to think and solve problems on their own, only giving them advice for a completed attempt. This can be compared to the grading and feedback phase in a traditional classroom course format, or the project review phase common to design projects.

A major distinction among these styles concerns motivation for, and timing of, teaching feedback.

3.1 Tutoring Research

Tutoring systems instantiate the classic theory of classroom teaching, that students should learn things in a stepwise fashion. These systems present a session based on an analysis previously made by a courseware designer of what a student needs to learn and in what sequence. Such an approach is not generally responsive to an individual user, except to indicate performance scores. Coaching and help systems respond immediately when the user has a problem. A critic does not provide suggestions until the user has completed a solution.

Current theories of instructional design (Reigeluth, 1983) focus on important issues of syllabus design (Aronson and Briggs, 1983), student motivation and teaching students how to learn (Collins and Stevens, 1983). Dennis Aronson and Leslie Briggs, for example, developed a widely referenced list of "instructional events" centered on steps of teaching a topic (Aronson and Briggs, 1983):

(1) gaining (the learner's) attention;
(2) informing the learner of the objective;
(3) stimulating recall of prerequisite learning;
(4) presenting the stimulus material;
(5) providing learning guidance;
(6) eliciting the performance;
(7) providing feedback about performance correctness;
(8) assessing the performance;
(9) enhancing retention and transfer.

Such a list encourages the teacher to consider the *teacher's* goals for the student and use these objectives to guide interaction with the student.

How should educational approaches themselves be judged? The educational goals differ in varying approaches. Glen Snelbecker (Snelbecker, 1983) created a list of nine educational issues that may be used to evaluate the priorities of a particular teaching model. The teaching model might emphasize the importance of student *preparation* for a topic or approach, gaining and keeping student *attention*, quality teaching *presentation*, timely *response*, appropriate *feedback*, teaching for *retention and use*, presenting material to aid student *understanding*, encouraging students to learn to use their *creativity*, or the model might focus on *management* of teaching situations. A teacher's goals may be met by choosing materials based on a model which emphasizes the desired aspects. With the exception of student preparation, the COACH teaching model's adaptive approach allows it to concentrate on all of the above issues as appropriate.

Tutoring systems depend on syllabi to guide students. Computer tutors most often guide lessons by branching on students' responses to yes/no, multiple choice, or word fill-in questions. Other *tutoring* work uses more sophisticated techniques to help guide a user through a lesson plan.

CAI work dates back to at least the late 1950s (Rath *et al.*, 1959). Suppes (Suppes, 1967) describes the classic CAI method of mechanizing a traditional paper textbook. In place of a textbook, a user reads text and questions from a computer screen. The user works through the text by typing word or number responses to problem questions. Most commercial *tutoring* systems use this mechanized programmed textbook method of presenting information to a student. Even early *tutoring* systems varied their responses relative to a user's answers, something that currently available commercial *help* systems fail to do.

CAI research has taken the syllabus approach to learning much farther than the initial automated textbook efforts. John Anderson's Lisp Tutor has a sophisticated way of presenting the lesson questions as programs for a student to write (Reiser *et al.*, 1985). Exercises require students to write programs designed to teach about a particular concept or tool. The student's solution can vary from the teacher's prototype in the naming of variables, but cannot vary the functions used; for example, a student cannot use the "IF" function where the system expects the "COND" function. The student answers questions and writes programs; the system guides the student through the syllabus. The system certifies a student as a learned programmer relative to the problems in the syllabus that have been completed correctly.

Anderson's system improves the learning abilities of Lisp students. His system expands on the CAI textbook-style syllabus. A student's progress is guided by knowledge formalized in "if, then" rules, and the word or number answers of early CAI systems are replaced with programs the user must write.

In one experiment (Corbett and Anderson, 1989), the Lisp Tutor was modified to allow users to use the Lisp interpreter to experiment with Lisp statements. Students could explore the Lisp environment if they desired to "try things out" without leaving the *tutoring* environment. Curiously, in this experiment, student progress was retarded by allowing them to work on things other than the solution to the current *tutoring* problem. In this otherwise controlled learning environment, the flexibility of allowing exploration seemed to distract the student (Corbett and Anderson, 1989). Educational settings in which students thrive on exploration do, however, exist.

Seminal work in applying AI in the field of education is typified by John Seely Brown and Richard Burton's productive collaboration. Brown and Burton's "Debuggy" (Burton, 1978) introduced knowledge representation and reasoning into CAI. In grade school studies, they showed Debuggy could teach a student about long subtraction with carrying, understanding the student's mistakes better than a teacher. Their approach to teaching subtraction included cataloguing the one hundred twenty or so possible types of mistakes a student can make while doing a subtraction problem. The system used a static sub-skill network to characterize what skills the student might be lacking which generated particular errors in an answer to a problem. For each possible mistake, the system had knowledge describing underlying missing concepts which could be responsible for the mistake. Debuggy's sophisticated representation of the problem domain enabled it to use a reasoning approach to evaluate bugs in student subtraction strategies. Pre-analyzing the entire solution and error domains gave the system the ability to explain all incorrect subtraction algorithms.

The reasoning approach which Brown and Burton used in Debuggy required them to *completely* describe and analyze all possible subtraction errors. Many domains of interest are much larger than subtraction; identifying all possible mistakes in them is usually impractical. In fact, Brown and Burton found teachers seldom understood subtraction in the detail that the Debuggy approach used to reason about potential gaps in student understanding.

The syllabus approach to teaching has been validated as useful in some situations. For example, the order in which students learn two kinds of loop constructs can determine how easily they can master them both. Jeff Bonar (Bonar and Soloway, 1985) ran a large scale study to test this. Specifically, he taught some students the `REPEATUNTIL` construct in Pascal before the `DOWHILE` construct and some after. The students did better when they learned `REPEATUNTIL` first. The order in which new skills are introduced can be important, even when the learning of one skill is not a prerequisite to the learning of another. Evaluations of the sort done by Jeff Bonar are especially instructive. Unfortunately, because various educators (and learners) may have different goals, not all issues of educational approach can be definitively resolved.

3.2 Help Systems

While *tutoring* systems present a curriculum through which a student travels, *help* systems at the other extreme allow motivated users to ask questions of the system. Many students are not motivated to follow the rigid lesson plan of a *tutoring* system. Many computer users come to an unfamiliar computer system with relevant experience and do not need to learn everything about it as though they were novice computer users. Their reason for using the new system may be that they have a particular task they want to perform that requires that system. Students absorb information when they have a use for it. Rather than provide a generic syllabus for learning the entire system, it is preferable to center the aid users will receive from the system on their specific needs related to that task.

Unlike *tutoring* systems, *help*, *coach*, and *critic* systems work with the students in productive situations. Systems that support student goals can allow the student flexibility. They can also provide user support more easily than systems that give students simplified so-called "training-wheels" *tutoring* outside of their work environment. Training-wheel *tutoring* systems protect students from a realistic work situation, but must be left behind when a student is ready to experiment or begin a real project. Systems which respond to user inquiries in work situations are termed *help* systems. Unlike other CAI research, most research on *help* systems has focused on the quality of information available to aid the user and has not extensively explored the use of AI or other strategies for delivering the information appropriate to the situation (Borenstein, 1985).

Nathaniel Borenstein (Borenstein, 1985) performed behavioral experiments showing that *help* systems are more effective when they are available from within (integrated in) the computer program than when a separate *help* system must be consulted. His studies also showed that *help* systems are improved when they give users context-dependent responses, basing the information users receive on the part of the computer program with which they are interacting. Borenstein also observed that the quality of *help* text and its relevance to the particular situation are more important than other usability issues, such as graphic design or the ease of asking for help. That is, content matters more than form.

3.3 Coaching Systems

A computer aid for using or learning a body of knowledge may be called a *coach* when, like a human coach, the computer trains, reprimands, gives aid for a personal weakness, and tries to provide a needed idea or fact when appropriate. As well as being extravagant, human coaches can be wrongly perceived. There is evidence that continuous human guidance provided during students' writing activity is often seen as ill-intended, leading students to reject it (Brehm and Brehm, 1981). Computer-based guidance does not arouse such a reaction (Zellermayer *et al.*, 1991). However, until now such *coaching* systems have relied on what users are doing (context) or whether they have made an error, without using any representation of the users' actual performance or users' understanding of the system.

To be able to respond to the user's actual level of proficiency, the system needs to learn from the user's actions. If a *coach* system had this ability, it could be said to have an adaptive user model (AUM). The system could build the adaptive model by asking a user questions. Elaine Rich's GRUNDY (Rich, 1983) system is well known for using a simple nonadaptive user model to choose books for users. A user filled out a "form" which her system used to create a user model. GRUNDY consulted this user model "stereotype" to select library books which might be interesting to that user. An adaptive version might include feedback and followup questions. This work, instead, explores user models which are built by watching the user's actions (Selker, 1989).

An important issue is whether unsolicited feedback is intrusive, derailing and frustrating users, or whether it can offer welcome advice (Bereiter and Scardamalia, 1984). Michael Zellermayer (Zellermayer *et al.*, 1991) performed a study which gave mixed results concerning unsolicited computer-presented advice. A system called "The Writing Partner" cued students with so-called "metacognitive" questions, concerned with higher level information than the writing itself, such as planning and organizing. The system attempted to help the students plan and organize their papers by asking questions such as: "Do you want your composition to persuade or describe?" In a comparison of three groups of students writing essays, one group received no guidance, one group received metacognitive guidance when they solicited it from "The Writing Partner", and one group received unsolicited metacognitive guidance from "The Writing Partner".

Many people have the intuition that unsolicited computer advice would intrude and slow a user down. And, indeed, while using "The Writing Partner" during the training period, the students in the group that received unsolicited advice took longer to accomplish their work and did not show an improved essay writing ability. While this might seem to corroborate the impeding, intrusive advisor hypothesis, those same students who had been continuously advised on the metacognitive aspects of their writing tasks were able in essays written two weeks later (on paper, not using "The Writing Partner") to write better essays than students in the other groups.

This provocative study shows how a *coach* offering unsolicited help can teach a person a new skill. Unfortunately, it also indicates that learning this new skill (essay organization and planning) with this type of unsolicited help has a cost. Happily, this study provides new evidence that unsolicited help can shorten rather than prolong a task.

The idea of an explicit model or expectation of a user's performance is not new. Burton and Brown's electrical circuit trouble-shooting learning environments (SOPHIE 1, 2 and 3) (Burton and Brown, 1982) give important results concerning user modelling. SOPHIE 3 included an evaluation strategy which compared students' performances in designing circuits with those of expert circuit designers. The system reasoned about differences between novices and experts; in so doing, it attributed problems encountered by the novices to bugs in the novice user's otherwise expert approach. SOPHIE research promoted user exploration of the domain as a way of improving the task relevance of a syllabus. Since either the system or the user could control the session, these systems can be said to have provided *mixed initiative* interaction.

One important conclusion which this work (and others e.g., Feldman, 1980; Genesereth, 1982) put forward was the fallacy of the novice having an expert user model with some missing parts. Burton and Brown, instead, concluded that novices have a qualitatively different model of a domain than experts. Because of this, a novice user cannot easily be evaluated relative to an expert model. An expert has understanding which does not necessarily follow the procedural and simplistic analysis of a beginner.

Educational systems like SOPHIE or Burton and Brown's WEST (Sleeman and Brown, 1982) involve users in a little world in which they can explore, learn, and try things out. Games with simulation are widely used in CAI (Sleeman and Brown, 1982). They are particularly appropriate in *coaching* systems. The feedback and integration of such environments is natural for the *coaching* paradigm. Game user interfaces often include a consistent simulated environment referred to as a microworld. Microworlds and other game teaching approaches have the advantage of addressing student motivation as an explicit goal.

3.4 Critic Systems

A *critic* system criticizes or evaluates work at a specific point, either when requested by a user or at the end of a session. Only when a user has come to this point does the *critic* system offer its aid. This approach has often been employed in order to allow the computer to analyze student work off-line.

The advantage of a *critic* system is that it gives the user time to reflect upon the system's suggestions and create a solution without interference. However, the disadvantage is that the advice is not available at the time the problems arise.

Adrian Zissos and Ian Witten (Zissos and Witten, 1985) built a prototype adaptive *critic* system which could analyze transcripts of EMACS text editor usage after a user session. ANACHIES, as it was called, could decide how to improve a person's use of EMACS editor commands. Zissos and Witten's paper offers a

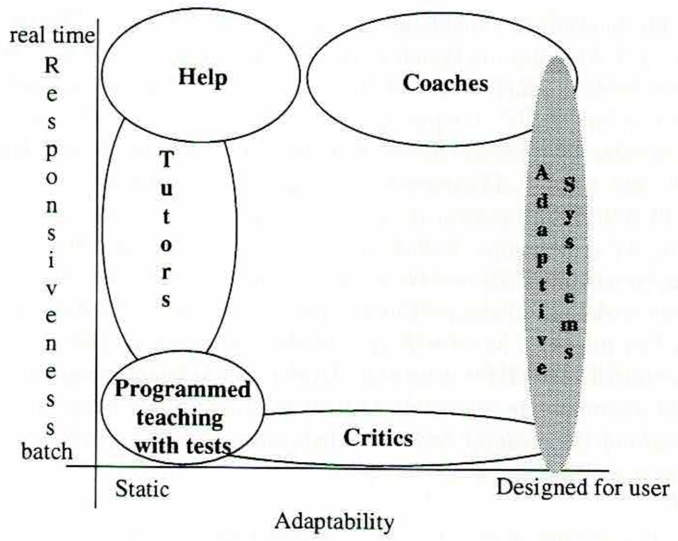


FIG. 6. An illustration of the responsiveness and educational goal differences which characterize different computer learning environments. The horizontal dimension, adaptability, refers to the system's ability to be changed for a situation or user. The shaded ellipse indicates where systems which automatically change or adapt to a user's goals would lie in the illustration.

pessimistic view of the feasibility of having a system react as the user needs assistance. Their research convinced them that the computational requirements for using adaptive AI techniques in interactive applications could not feasibly be met with the computers available in the foreseeable future, a cynicism which this research demonstrates to have been unwarranted.

This section has reviewed CAI teaching styles in terms of their responsiveness to users and how their educational goals are chosen (Fig. 6). *Critic* systems, for example, offer batch responses, while *coaching* environments respond interactively to a user.

While much work has been done on *tutoring* environments, demonstrations of real-time adaptive teaching environments have not been convincing. Researchers still question the reasonableness of real-time adaptation. Until now, neither the utility of adaptive interfaces of any type nor the possibility of unintrusive unsolicited help have been shown. COACH offers results addressing these concerns. The following sections describe the COACH interface, concentrating on student-motivated teaching and learning interaction.

4. Requirements for an Adaptive Help Testbed

Tools for building computer aided instruction systems (CAIs) are usually referred to as CAI authoring systems. Tools for managing a rule base and providing a ready-made production system that can run these rules are often referred to as expert system "shells", connoting their ability to be hard containers that can store knowledge or representations (Barr and Feigenbaum, 1984). The earliest described rule system, PLANNER (Hewitt, 1972), with its implementation, MICRO-PLANNER (Sussman *et al.*, 1970), could be described as a shell for developing AI applications. It was quite a general system in which "theorems", which can be generally thought of as rules, could be specified as being useful for forward or backward chaining. Filters specified classes, which can be loosely thought of as rule sets. More widely available systems like EMYCIN (Clancy, 1986) and Intellicorp's KEE (Fikes and Keeler, 1985) contain many tools to represent and reason about a domain. As well as providing an authoring system in which teaching information could be changed, an adaptive user model would be used as a shell in which reasoning about the teaching process itself could be modified.

An AI practitioner using an expert system shell can utilize the shell's representation and reasoning machinery without having to build it from scratch. Shells are designed to lever AI practitioners' efforts by allowing them to create expert systems by merely describing the rules relevant to the task or skill domain; the practitioner may then use the reasoning tools provided in the shell. The challenge to building an AI application is understanding the domain knowledge that is to be embodied in the application, understanding the reasoning relationships in the knowledge, formalizing these, and converting them into the AI shell's representation and reasoning formalisms.

COACH, which is described in detail in the following sections, is a shell for adaptive coaching. It provides machinery for formalizing both teaching and domain knowledge for coaching user interfaces. COACH is designed to allow a curriculum designer to encode a domain to be taught. It is also designed to allow an educational researcher to encode theories of when and how to present information to a student. Additionally, COACH allows the cognitive scientist to encode approaches to gathering user information and methods for altering treatment of students based on their responding behavior.

The courseware developer's process of converting the system to teach in a new skill domain requires the following steps:

(1) Identify a task or skill domain.
(2) Identify any delimiters and other token types that the system does not already have.[1]
(3) Write token handling function methods for the domain's token types not already supported.[1]
(4) Change the token table for parsing delimiters.[1]
(5) Identify commands in the skill domain for which help initially will be made available.
(6) Describe the syntax of the language being taught in the COACH formalism.[1]

A primitive help system would then exist for the skill domain. The system would be able to check user syntax, look for undefined functions and variables, learn about user examples and add help for new and system functions. It could also increase and decrease help and change levels of assistance.

The system would not yet know about relationships between syntactic units, concepts, basis sets or required knowledge. As described in Section 6, these create a representation for generating adaptive help which can remind a user of related material and concepts in the skill domain when appropriate. Identifying the relationships between parts of the domain then allows the developer to add deeper knowledge about the skill domain. This can be accomplished with the following steps:

(1) Identify skill domain concepts.
(2) Identify basis sets in the skill domain.
(3) Identify required knowledge for skill domain parts.

[1] COACH/2 automates much of 2, 3, 4, and 6 above, allowing the author to identify and annotate a GUI uing a WYSIWYG authoring tool.

(4) Write description, syntax and example text for different expertise levels of each skill domain part.

Just as AI shells have simplified AI application development, a shell for testing and extending adaptive help simplifies experimentation and development of adaptive help applications. As an expert system shell requires a practitioner to formally understand the reasoning that is included in an AI system, so an adaptive help shell requires the courseware designer to understand the formal syntax of the language for which the system is to produce help and the relationships between its parts (Selker, 1989).

For as long as computers have existed, people have talked of the possible value of intelligent computer assistants. Unfortunately, the architectures suggested have not been entirely successful. Either they cannot run interactively with a user, or they have not been shown to improve the user's performance (Zissos and Witten, 1985; Gentner, 1986; Waters, 1982). The COACH implementation has neither of those problems; it has been demonstrated to run interactively and improve user performance.

5. Theoretical Considerations For Creating COACH

After years of work, researchers came to the conclusion that proactive interactive adaptive computer interfaces were not feasible (Zissos and Witten, 1985; Gentner, 1986; Waters, 1982). This work challenges that conclusion and demonstrates an example of such a system. The demonstration relies crucially on an understanding of the constraints and requirements of real time response. To satisfy these constraints and requirements, the following questions must be addressed. What kinds of domains are interesting and possible to work with? What kinds of errors do users make? In what ways can these errors be addressed? What teaching techniques can be used without undue computational overhead? This section addresses these issues. Section 6, which describes the COACH architecture, addresses language theory issues that pertain to evaluating user work (see Section 6.4).

5.1 Suitable Domains for Demonstrating Adaptive User Help

Demonstrating that adaptive user help is a viable approach requires showing both that it will work for an important class of interfaces and that it will provide valuable improvements over current help systems. The class of interface chosen was text; the first domain chosen was Lisp.

Text entry is probably the most common interactive technique used to communicate with computers. Even in the age of graphical languages, many operating system command languages, computer application interfaces, text markup languages and programming languages are interpreted text interfaces. Text was chosen for the initial COACH implementation because text is computer parsable and used in so many interfaces. Also, learning a text-based interface forces a user to confront difficult educational issues, and to work with incomplete and inconsistent knowledge.

Computer interfaces often include commands which are too complicated or too seldom used to be easily remembered. To show that COACH will alleviate such problems, the representative demonstration domain should contain many commands with complex syntax.

Computer interfaces contain many interdependent commands and concepts. Thus, a representative demonstration domain should also include complex interdependencies.

Computer interfaces often allow more than one way of doing things. Therefore, a demonstration domain should permit redundancies and alternative solutions.

The information the users must master often changes as time goes on. Therefore, a demonstration domain should be extensible.

Educational domains are often too large to enumerate or analyze fully. The domain should be larger than the implementation can fully represent.

To demonstrate adaptive user help, a domain was sought which would show several strengths of the approach: the ability to cope with a large and changing domain, the ability to extend help to include additions made to the domain by the user, the ability to permit solutions to have a complex structure, and the ability to coach in domains relying upon many difficult concepts.

Lisp programming is a domain which provides all of these challenges. Many pedagogical tools are designed for limited or toy domains. Lisp is extensible, requiring a help system that will work even when the skill domain changes and enlarges. Working with a domain that changes and enlarges (i.e., an "open system" (Hewitt, 1985)) is a test of the robustness of COACH. Lisp is a complex open system that allows demonstration coaches to work in a realistic domain. Lisp has redundant ways of doing things that require the system to act in ambiguous situations. Lisp is also a domain for which other people have built intelligent tutoring tools. This allows COACH to be compared to and benchmarked against their work. COACH/2 was explicitly designed to allow experimentation with teaching GUI procedures and coaching the use of dialog boxes.

5.2 Classifying Knowledge Deficits

A user is trying to learn a skill domain, e.g., Lisp. The initial implementation of COACH was designed for skill domains that require interpreted text input. Users must remember keywords, delimiters, syntax, and their previous input to effectively use text-based programming languages and computer command line interfaces. People forget; even experts are always working with gaps in their knowledge. These gaps may be classified into three types:

- *Issue*—the focus of what a person is trying to learn. This is the material users are aware of and *know they do not know*. On-line help system queries are useful for learning *issues*. COACH shows such information automatically.
- *Incomplete*—the knowledge a person *does not know exists*. These are actual holes in the particular user's knowledge. COACH points out general knowledge when a user is having difficulty in a particular area.
- *Inconsistent*—the knowledge users *think they know but do not*. COACH points out errors to highlight such inconsistencies.

5.3 A Classification for the Help to Provide to Users

Interesting models and theories of instruction and instructional design exist (Reigeluth, 1983). David Merrill's (Merrill, 1983) "Component Display Theory" describes structures which educators use to organize their efforts. He separates domains of content into facts, concepts, procedures, and principles which can be known well enough to remember and apply correctly in appropriate situations. Such a taxonomy centers the process of learning on a domain and its interrelationships. This is extremely useful for creating a network of knowledge relationships which characterize the domain.

Edwina Rissland (Rissland, 1978) created a simple taxonomy of help examples. It highlights the importance of providing different kinds of examples appropriate to the expertise level of the user.

Rissland's help taxonomy consists of four levels of help:

- *Starter* help is used at a novice level. Only simplified basic information is provided. Novices depend on the literal cues in a problem situation (Glaser, 1985). The information given them, then, must be carefully designed so as not to mislead them.
- *Reference* help is more complete to familiarize users with standard usage.
- *Model* help is a complete description of what something is and how to use it.
- *Expert* help is machine-level description such as one might find in reference manuals.

This taxonomy of help examples shown to a user can be refined to segment each *level* of help into types of help a student may need. For COACH, this taxonomy is extended to distinguish and include examples of correct usage, legal syntax and descriptive text telling how and when to use something:

- *Example* help is an actual demonstration of an exemplary solution or solutions. Despite efforts to teach design through concepts and theory, the only effective teaching tool for design is commonly agreed to be the providing of examples (Vertelney *et al.*, 1991). Moreover, procedural and syntactic knowledge are often most easily conveyed through examples.

- *Description* help is an explanation of the utility and use of a solution type. This information can range from philosophical background to an explanation of the use of a specific language part suggested for the solution to a user's problem.
- *Syntax* help is a template showing the structure of a legal solution. For users to apply a specific statement in varying situations, they must internalize a model of its utility; syntax is the essence of a concise definition.

5.4 Tracking User Proficiency as the User is Working

Various approaches to analyzing knowledge about users have been tried and have been found to be problematic. Systems such as Don Gentner's (Gentner, 1986) used mathematical proofs of programming correctness to decide what a user was doing. The problem with this approach was that the computation required for these proofs grew exponentially with the amount of user work under analysis (Hopcroft and Ullman, 1979) which limited its utility for large bodies of work.

Zissos and Witten's EMACS critic system, ANACHIES, used cluster analysis to make determinations of user capabilities (Zissos and Witten, 1985). This, too, was a computationally intensive procedure which grew exponentially with the size of the language.

Care has to be taken to create an architecture capable of recording user activity in a representation which it can use to reason about how to provide help and react as the user is typing.

Careful use of representational, reasoning and learning techniques make this real time response possible. Several strategies can be used in the representation to limit knowledge search and access problems. Relationships in the knowledge representation may be recorded and stored as links as soon as they are known. Search would thereby be diminished by the pre-defined network created by these links. By limiting the depth of relationship links, search difficulty caused by complex links is decreased. As many user model characteristics as possible would be recorded as scalars, so as to limit representation growth and reasoning difficulty.

Several strategies would be used to make the reasoning system efficient. The rule system would use an "if/then forward-chaining approach" and avoid the less determinate, computationally extravagant "backward-chaining means/ends analysis" (Barr and Feigenbaum, 1984). The reasoning can be broken into rule sets which would be used on specific knowledge and on specific parts of the reasoning process. These segmentations of reasoning problems decrease complexity when searching through the rules and searching in the knowledge.

Finally, it is important to choose learning techniques which are feasible for real time computation. In order to learn in real time, the system should limit itself to opportunistic and simple hill-climbing learning. Below is a classification of learning taken from *Machine Learning* (Michalski *et al.*, 1983). This classification is annotated to show low computational cost learning techniques which could be used in an adaptive learning system to organize the ways the system can change its behavior:

- *Learning from examples* is the practice of using specific solutions already achieved in more complex situations. This technique can be used in the COACH interaction style to collect user-provided syntax and examples, and to offer help for user-defined variables and statements. In the interaction style, syntax may be collected by recording user definitions. Examples can be collected from user work.
- *Learning by analogy* is gathering knowledge in one situation for use in another similar situation. This kind of learning can be used by COACH to decide when to explain things in terms of skill domain parts the user already knows. The help system architecture would also include a network of skill domain parts in which it could access information for this purpose.
- *Learning from instruction* is utilizing a user interface or special language to introduce knowledge into the system. A courseware designer uses this technique to modify a CAI system without programming. Expert systems and most state-of-the-art AI education systems rely exclusively on developer modification to change response behavior. Modifications and improvements for the Lisp Tutor (Corbett and Anderson, 1989), and the Lisp Critic (Fischer *et al.*, 1985) are made in this way. COACH is designed to allow a researcher to add facts and rules that improve the adaptive user model system without writing Lisp code. Observations of students using the system give the researcher ideas of how to change the way the system treats a user in different situations. These ideas are put into additions and changes in presentation text or presentation rules.
- *Learning by programming* is simply the practice of having a developer add knowledge to a system by actually writing code. Before rule systems existed, this was the only way of improving an AI application. Any system can be extended by programming to add function or change domain.

The technology described above is used in the following section to present the COACH architecture, which enables real-time adaptive help in interactive computer environments.

6. An Architecture for Adaptive User Help

This section introduces the structure that enables adaptive coaching.

The adaptive help system can be modeled with four interacting parts or objects: