User Modelling in Cooperative Knowledge-based Systems

Thomas W. Mastaglio

CU-CS-486-90 August 1990

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80399-0430

Author's Current Address:

LTC Thomas Mastaglio
P.O Box 298
Fort Monroe, VA 23651
mastaglt%mon1@leav-emh.army.mil

Table of Contents

List of Figures

User Modelling in Cooperative Knowledge-based Systems

Thomas W. Mastaglio
Department of Computer Science
University of Colorado, Boulder, Colorado 80309

Abstract

Theoretical studies and implementations of computer-based critiquing systems indicate that it is desirable to enhance the approach to better support human-computer collaborative effort. A user model will enable these systems to individualize explanations of their advice to provide better support for cooperative problem solving and enhance user learning. User modelling research in advice-giving dialog and intelligent educational systems was studied together with theoretical analyses of the limitations of human-computer interaction, and empirical observations of human-to-human collaborative effort.

A framework for a user modelling component for a critiquing system was developed and implemented in a critic for LISP programs. The user models developed by the system were compared to self-assessment questionnaires completed by subjects learning the LISP language. The analyses indicated a favorable correlation as well as potential improvements to the framework. The user model is based on the conceptual domain model required for explanations; its semantic structure allows the system to implicitly enrich the user model contents. This work provides a framework for a user modelling component that can support a more general class of cooperative knowledge-based systems.

1. Introduction and Theoretical Background

1.1 Introduction and Overview

This report presents a user modelling approach to support cooperative problem solving. The problem is to represent, acquire and provide access to individual user models to support computer critics. Critics are knowledge-based computer systems that use the critiquing approach to support users in their work. Critiquing enhances the users' work (these are called performance critics) and supports their learning (called educational critics) [Fischer et al. 90]. Future systems that support users in their working environments need to accomplish both objectives. A user model will be an important component of such a system; it will assist the system give knowledge-based advice and explain that advice. A framework for this type of user model was developed, implemented, and evaluated. Other user modelling research was studied as were approaches for generating explanations of domain expertise. Where possible, proven techniques from other user modelling research were incorporated. An understanding of what is required to generate explanations guided the architecture for a user modelling component that is an extension of an existing critic for program enhancement (LISP-CRITIC).

The approach combines methodologies developed by other research with innovative acquisition techniques. Major contributions are a framework for a user modelling component that can be used in other applications and a set of techniques for indirect implicit acquisition of the user model. The latter techniques use the semantic structure of a conceptual domain model. The framework can support a broader range of applications, and systems that use other interaction paradigms. The user model captures individual expertise at the conceptual level of the application domain in a fine grained representation of users' knowledge.

1.2 Cooperative Problem Solving

There are methods and technologies in Artificial Intelligence that can improve computer users' productivity. A paradigm for designing systems that goes beyond current autonomous expert systems addressing the needs of humans and their potential is that of *Cooperative Problem Solving Systems* [Fischer 90].

In cooperative problem solving, the user and computer-based system work on the same problem using a collaborative interaction style. Cooperative systems allow the combining of human skills and computing power to accomplish a task which could not be done by either the human or the computer alone; or in those cases where it could be, the quality of the result or the speed with which a solution is obtained is significantly improved when the two agents work together. The idea of symbiotic cooperation between the user and the computer has also been embraced by researchers in the related field of decision support systems [Mili, Manheim 88; Manheim, Srivastava, Vlahos, Hsu, Jones 90; Hefley 90]. To achieve a cooperative system requires the system to adapt to individual users; a system can adapt successfully when it knows something about the individual; and user modelling provides the system the capability to acquire and use that type of knowledge.

1.2.1 Cooperative Problem Solving versus Expert Systems

Autonomous expert systems differ from cooperative problem solving systems in their approach — they develop problem solutions independent of user input. In many problem solving situations, articulation of the task is difficult and people are unsure of their objective or exact problem. They start with a general view of what they expect to achieve and refine their own understanding and problem specification during the solution process. To put cooperative problem solving systems into the context of current artificial intelligence we need to consider system designs that go beyond the current notion of expert systems.

The major difference between classical expert systems (such as R1 or MYCIN) and cooperative problem solving systems is that the users of cooperative problem solving systems are active agents. They are actively engaged in reasoning about the problem and generating the solution rather than participating as mere providers of information to the system. Conversely, traditional expert systems ask users for information about the problem situation and then return a solution; from an operational perspective they function as a "black box." Cooperative problem solving system are designed so that the user and the system share in the problem solving and the decision making. Because human-computer communication is central, cooperative problem solving systems require better interaction facilities.

1.2.2 Communications Requirements for Cooperative Problem Solving

Communications between a human and a computer is a fundamental design problem for cooperative systems. Specifically, to facilitate interaction between a human and a computer it is necessary to exploit the asymmetry of the two communication partners. Each agent or partner contributes what they can do best. People are better than computers at applying common sense, defining goals, and decomposing problems. Computers should be used as an external memory, to do consistency checking, to hide but not lose irrelevant information, to capture and summarize problem solution steps, and to help visualize concepts.

Communicating requires the human or computer agent to know or assume something about its partner. One approach is for systems to capture implicit assumptions about all of the users in their design (a default generic user model). Idealistic systems might adapt everything they do for each individual. A more reasonable middle ground, is for systems to tailor their side of an interaction based on what they are able to infer about their human partner with user modelling. Understanding what is needed to accomplish user modelling requires an examination of the human-computer communication process.

When two agents are engaged in cooperative effort, a process of natural communication takes place. Natural communication is more than natural language; it is the ability to engage in a dialog that makes use of diectic techniques, indexicals, graphic representation, and references to previous conversation. It is important to think in terms of the broader context of natural communications rather than natural language because systems that use natural language techniques are frequently too brittle [Winograd, Flores 86]. They experience breakdowns in interacting with users when the unexpected occurs, especially in situations not anticipated by the system designer; techniques are needed to overcome these breakdowns.

Computers can understand task domain knowledge; use it as the basis for advice, and as a source for guidance about how to better communicate that advice and explain it. The distinction between advising users (telling them what to do) and explaining that advice (telling them the reason for it) is important. The distinction is significant here because LISP-CRITIC gives both suggestions (or advice) and we endow it with the ability to explain those suggestions. The latter process makes primary use of the content of the user model.

Even with good user models, cooperative problem solving systems will not always be successful in their initial attempt to provide advice or explanation — what is sometimes called a "one-shot" approach to the interaction. Even people do not always "get it right" the first time; a great deal of the effort in any communications involves repairing breakdowns between the two partners. Techniques to achieve something similar are needed in interactive systems. When users do not understand the system's advice, critique, or explanation, followup techniques are required [Moore 89].

In addition to helping users to solve their problems, systems should also help them learn about the task domain. Systems that serve knowledge workers, such as designers, authors, and programmers, must accomplish both objectives. During normal use it is difficult to distinguish a situation or episode oriented strictly on problem solving; learning and doing frequently intermix during human-computer interactions.

1.3 Learning Environments

Computer-based learning environment enable users to improve their proficiency in a domain by providing for the knowledge communication process [Wenger 87]. One approach to designing such learning environments is to build instructionally focused systems, such as intelligent tutors [Sleeman, Brown 82]. In many situations it is desirable to provide a learning opportunity within the context of a user's work and with the user in control of the interaction. Here, paradigms that are more general than intelligent tutoring systems are needed. There is interest in and research on the problems involved in providing learning environments in a variety disciplines: cognitive science [Burton, Brown, Fischer 84; Fox 88], human-computer interaction [Fischer 88a], and computer-based training technology [Duchastel 88; Mastaglio 90a].

1.3.1 Foundations for Learning Environments

There is a need to find efficient and practical ways to improve education using computer technology. Studies by Bloom and colleagues demonstrated that if we can individualize instruction, significant increases in student performance can be reasonably expected [Bloom 84]. A shortcoming of early computer learning systems was their fundamental design philosophy; it was based on conventional ideas about programmed instruction used for self-paced learning material. An alternative approach using artificial intelligence techniques was first proposed by Carbonell [Carbonell 70]. During the ensuing 20 years since that work, research efforts have resulted in the development of several paradigms for computer-based instruction using artificial intelligence [Wenger 87].

The computer can provide a suitable context for learning both procedural and declarative knowledge and as reasonably priced hardware support for graphical displays becomes a reality, these types of environments will become more feasible. The theoretical limitation to effectiveness is not the availability of material or simulated scenarios, but incorporating, into the system, a didactic agent to guide learners. For learning to occur in a computational context in which users are involved in some action, the system must provide feedback to its users in the form of advice, critiques, and explanations; higher quality feedback provides for improved learning.

One way for a system to provide high quality feedback is to use knowledge-based components. Three of the processes that take place in learning environments require knowledge:

1. providing information or instruction with an expected outcome that users' knowledge improves,
2. determining the state of users' present knowledge (user or student modelling), and
3. motivating the user to learn.

To execute the first process, a computer agent has to know both the domain and strategies for guiding users' learning. Methods to accomplish the second process are the subject of this research. Cooperative problem solving systems assume that the third process is inherent in the situation,that users are motivated because they have chosen to use the system in the first place.

There are three distinguishable paradigms for computer-based education based on artificial intelligence techniques — three types of intelligent learning environments [Mastaglio 90a]:

- Intelligent tutoring systems . [Sleeman, Brown 82; Psotka, Massey, Mutter 88a; Polson, Richardson 88].
- The coaching approach [Burton, Brown 82].
- The computer-based critic [Fischer, Lemke, Mastaglio, Morch 90].

Here, critiquing is emphasized because it holds promise as a general approach. A computer-based critic can help users improve their skills within their working context, but besides giving suggestions critics need to be able explain their expertise and not just in a canonical form but in a manner that is tailored to each user's current state of knowledge. For critics to fully support the learning process they need explanation and user modelling capabilities.

Because critiquing is not the only approach to designing a learning environment the user model was developed to be able to serve a general class of learning environments. The notions of situated learning [Brown, Burton, Kleer 82], and learning on demand [Fischer 88a] provided general guidance and understanding.

1.3.2 Learning on Demand

Learning on demand supports learning in the context of a user's work, allowing people to improve their knowledge when a need arises [Fischer 87]. It is based on an optimistic view that people want to know how to do their jobs better and are willing to engage in learning activities. If computer-based working environments, (e.g., design environments [Lemke 89]) are to provide a complete and appealing context for working, then they need to support learning on demand. Learning environments can be designed as extensions of existing computer-based systems, ones that already support some types of work. Some examples of these are design, programming, and authoring.

To support learning on demand requires architectures that differ from instructionally oriented systems. Learning on demand requires that the user retain primary control of the learning situation. A continuum of approaches to learning environment design is shown in Figure 1-1; the dimension for the continuum is control of the interaction. At the left extreme, primary control is the responsibility of the system; at the right, the responsibility of the user. In the center, the system and the user share responsibility for control.

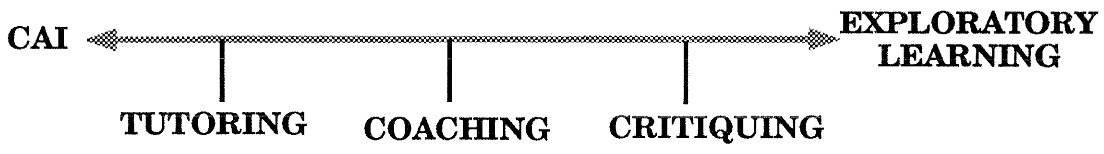


Figure 1-1: A Continuum of Approaches to Learning Environments

In the middle of the spectrum are the three knowledge-based paradigms for learning environments. Within the class of intelligent tutoring systems various degrees of control may be given to the student but, in general, problem selection, monitoring of student actions, and intervention fall under the auspices of the intelligent tutor. Coaching, as used in the WEST system [Burton, Brown 82], allows users to practice skills in a computer-supported context with a computer-based intelligent mentor "watching over their shoulder." The coach intervenes with suggestions and instruction when appropriate.

Research efforts in both exploratory learning [Miller 79] and coaching systems [Brown, Burton, Kleer 82] recognized the need for a paradigm which is not as intrusive as a coach but extends the power of exploratory environments by providing the user contextual, on call, intelligent assistance in the application domain. Computer critiquing is a paradigm which meets that requirement, it allows users more control of the interaction as well as responsibility for selecting the problem. A system that provides for learning on demand will situate learning in the user's context, it is in this context that users request support for learning, understanding the situatedness of the working and learning context is important.

1.3.3 Situated Action

Suchman made an in-depth study of how people interact with machines [Suchman 87] and argues that research approaches which attempt to explicitly represent or infer user plans are inadequate. First we need to explore the relationship of knowledge to action, keeping in mind that a machine has fewer resources for interpreting the user's behavior than a human.

A cautious attitude toward what to expect from a machine is important because during face-to-face communication between people there are resources that help them detect and remedy trouble when it develops, for example, facial expression or tone of voice. The same range of resources are lacking, for the most part, in human-machine interaction because of an impoverished communication channel. The orientation in this project was on communications capabilities which are within current limits of technology, pursuing how those capabilities could best be used to improve what the systems knows about the user, and also what domain knowledge is required by the system. One such idea considers the content of the man-machine interaction dialog as a source of information that can aid in acquiring models of users.

Any environment that provides users the opportunity to learn in the context of the task domain or by using a simulation of that context, requires that the system react so as to increase their understanding of the situated action [Brown, Burton 86]. Critiquing facilitates situated learning because it provides for learning in the work context. Users learn a set of conditions under which their knowledge can be applied and therefore enhance their understanding. This context, which is a central notion in situated learning, comes about naturally in critiquing systems. Critiquing combined with explanation approaches can clarify understanding and help to restructure users' knowledge [Psotka, Massey, Mutter 88b]. Learning has been traditionally supported with instruction, but a more likely situation, one similar to the manner in which human-to-human interaction often occurs, is to support it with an explanation capability [Wenger 87].

1.4 Critiquing

Critiquing, as a building systems technique, is of research interest in both artificial intelligence and human-computer interaction [Fischer, Mastaglio 89; Fischer, Lemke, Mastaglio, Morch 90; Mastaglio 89]. Critiquing is a way to use knowledge-based system techniques in situations where autonomous expert systems are inappropriate. In order for a critic to meet the goals of cooperative problem solving and accommodate user learning it needs to be able to explain system knowledge in an individualized fashion.

1.4.1 Foundations for Critiquing

We are interested in systems which support both problem solving and learning; critiquing is a paradigm founded on both of these concepts.

Cooperative Problem Solving. Previous critics, by their nature, operate in a cooperative manner; but can be further enhanced to more fully achieve the objective of a cooperative problem solving system. For users to accept critics as a useful feature of their working environment they need to provide explanations and, where appropriate, suggest alternative solutions.

Some shortcomings of traditional expert systems were already pointed out; another one is that these systems are inadequate when it is difficult to capture all domain knowledge. Because expert systems often leave the human out of the process, they require comprehensive knowledge that covers all aspects of the tasks; all "intelligent" decisions are made by the computer. Some domains are not sufficiently well understood, and to create a complete set of principles that capture them is not possible. Critics are suited to these situations because they need not be complete domain experts but can still offer helpful guidance even when their expertise is limited to only some aspects of the problem.

The traditional expert system approach is also inappropriate for ill-defined problems. This is because problems cannot always be precisely specified before a tentative solution is attempted. In contrast, critics are able to function with only a partial task understanding.

Support for Learning. The computational power of high functionality computer systems can provide qualitatively new learning environments; future learning technologies will be multi-faceted and support a portion of the spectrum of approaches shown in Figure 1-1. Some versions of intelligent tutoring systems developed in research laboratories allow the student to exercise greater control of the interaction. LISP TUTOR was reimplemented in a mode that permitted students to decide when to allow the system to evaluate their work [Anderson, Conrad, Corbett 89]. Student performance on post tests were equivalent for the immediate feedback (tutor-controlled) and the demand feedback (user-controlled) versions. Students actually took longer to solve problems when feedback was under their control rather than the systems, however, the quality of the learning experience is not degraded. This has clear implications for systems designed to support learning in situations where it is necessary for users to provide the problem specification. There is a recognizable trend for learning systems to move toward the middle of the continuum — closer to the critiquing paradigm.

1.4.2 The Critiquing Approach

Human-to-human critiquing is used in many problem solving contexts: design, authoring, student work groups, and collaborative research. People working together in these and similar areas naturally use critiquing as an interaction style. Critiquing is a way to present a reasoned opinion about a product (see Figure 1-2) such as a computer program, a kitchen design, a medical treatment plan; or an action, such as a sequence of keystrokes that corrects a mistake in a word processor document or a sequence of operating system commands. An agent (human or machine) that is capable of critiquing in this sense can be called a critic. Critics can be implemented as a set of rules or specialists for the different issues that may be associated with a product; sometimes critics are the term used for each individual system component that reasons about a single issue. In this project we call the entire system a critic; part of its structure is a composite rule set.

Critics do not directly solve users' problems, but they recognize deficiencies in a product and communicate those deficiencies to the users. Critics point out errors and suboptimal conditions that might otherwise remain undetected; frequently they suggest how to improve the product. Users apply this information to fix the problems, seek additional advice, or trigger requests for explanations.

Advisors [Carroll, McKendree 87] perform a similar function, but they are the primary solution source. Users describe their problem, and the computer advisor proposes a solution. Advisors do not require users to generate either partial or complete problem solutions . Advising as an interaction approach is best suited to situations where one-time advice is needed. User models are not as significant in these one-shot affairs, and ones that are used emphasize modelling users' goals rather than their domain expertise.

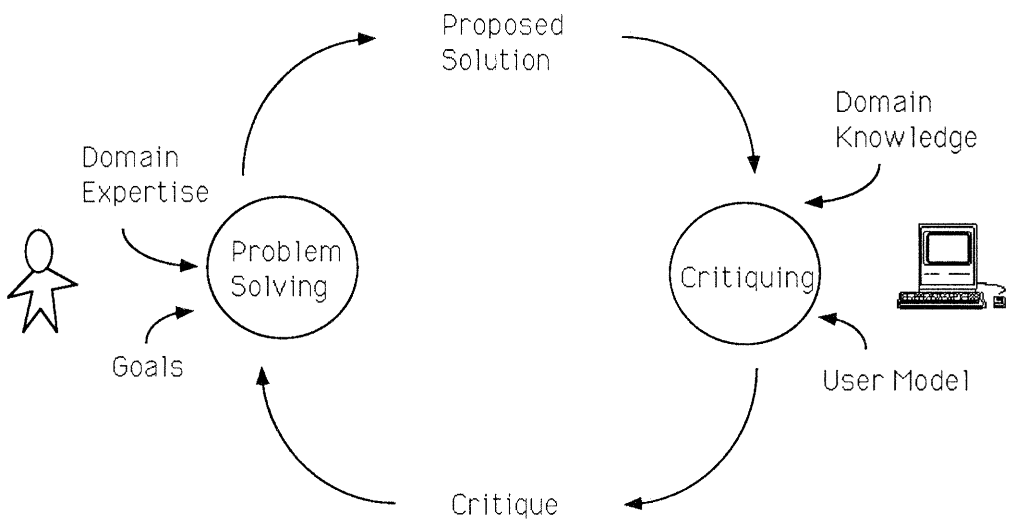


Figure 1-2: The Critiquing Approach

This figure shows that a critiquing system has two agents, a computer and a user, working in cooperation. Both agents contribute what they know about the domain to help solve some problem. The human's primary role is to generate and modify solutions, while the computer's role is to analyze those solutions, producing a critique for the human to apply during the next iteration of this process.

2. User Modelling

This section examines related research in user modelling and describes a general framework for the user modelling component of a system. Two related research areas have attempted to integrate idiosyncratic models of users. Research in Intelligent Computer Aided Instruction (ICAI) systems, most often referred to as Intelligent Tutoring Systems or ITS, use models of their students to guide instructional interaction [VanLehn 88]. Artificial intelligence techniques provide a basis for modelling users of advice giving dialog systems [Kobsa, Wahlster 89]. Work on user modelling in these areas provides a foundation for a user modelling framework to support cooperative problem solving.

2.1 An Overview of User Modelling Research

A survey of user modelling definitions together with an effort to correlate that research in both human-computer interaction and intelligent tutoring resulted in a useful taxonomy based on who owns the model and its function [Murray 88]. We conducted our own study in order to understand other research at the level of implementation methodologies and to determine what techniques could be used in the user modelling component of a critiquing system.

2.1.1 Student Models in Intelligent CAI

User Models in intelligent tutoring systems, called student models, have been the subject of ongoing research for a decade [Sleeman, Brown 82; Wenger 87; Polson, Richardson 88; Psotka, Massey, Mutter 88a; VanLehn 88]. Student models are derived from system knowledge such as rules, concepts, or strategies for learning a skill. The user's knowledge state is represented as a perturbation of that domain model — popular approaches are overlays representing the portion of the knowledge base a student knows, and a bug models representing user misconceptions about the domain. Differential modelling is the term often used for these techniques [Wilkins, Clancey, Buchanan 88].

The WEST project pioneered the differential modelling approach [Wenger 87]. Two ideas developed in WEST were used in this research. One is the idea that students' actions in the ongoing dialog with the system contain information that can be used to analyze the state of their knowledge. Another is the notion that knowing this state provides a mechanism to guide the computer coach in presenting new knowledge.

The genetic graphs approach was first developed for the WUSOR-II computer coach as a way to overlay domain knowledge with a learner-oriented linkage of rules, the latter represented as nodes in a graph model [Goldstein 82]. In another project, the genetic graph approach was used as a basis for modelling procedural skills in two quite different domains, one mental, subtraction, and the other motor, ballet [Brech, Jones 88]. Genetic graphs are normative models that define in their link structure the manner in which knowledge in a specific domain can be acquired by a student. This is an inherent limitation because the graphs have to explicitly capture all possible ways for a user to learn a domain entity.

It is is significant that ICAI systems can solve any problem on which their users (the students) will work because they must restrict students to those problems. This allows the system to do more detailed and specific model inferencing techniques than those available to systems that serve more open-ended problem solving situations. The requirement in our systems for more generality means that many techniques used in student modelling are often not robust enough to support real world problem solving. The problem-space limitations that are imposed by tutoring systems are what make them effective at teaching within those restrictions, and also what enables them to compile accurate and complete models of students. As an example, PROUST, is able to infer possible programmer plans for solving the single problem it uses in all instructional episodes, computing average rainfall with a PASCAL program [Johnson, Soloway 84].

2.1.2 User Modelling in Computer Advisory Systems

User models for advice giving systems based on natural language dialog have approached the user modelling problem by using artificial intelligence and linguistics theory. A popular approach is stereotyping; it was first proposed by Rich in the GRUNDY system [Rich 79]. Systems that use stereotypes acquire some characteristics of a user. When the system obtains sufficient information about users, it categorizes them as fitting a prestored stereotype, and the stereotype indirectly provides additional characteristics. One techniques, used in the work on GRUNDY, is to explicitly ask users for some of these characteristics. A user-generated description aids the system in selecting an appropriate stereotype.

Finin and Kass extended the stereotyping approach within a user modelling shell based on a hierarchy of prestored stereotypes [Kass, Finin 88a]. Their systems analyze communications between the user and the system using implicature rules adapted from Grice's Maxims for cooperative communication [Kass 87a]. An example of such a rule is *If a user says P, the user modelling module can assume the user believes that P, in its entirety, was used in reasoning about the current goal or goals of the interaction.* These rules, in conjunction with the stereotypes, infer a model of the user's goals and beliefs. Chin's work in KNOME, a user modelling component for UNIX CONSULTANT, used a double stereotyping technique, one for grouping domain concepts and the other for classifying a user's expertise. The stereotyping approach is useful for *one-shot* advisory type systems that need a quick approximation of the user in order to generate a piece of advice; it can also be used as a way to initialize user models.

Wahlster and Kobsa also use the content of a dialog to acquire a model of the user's beliefs, plans, and goals [Wahlster, Kobsa 88]. Their work attempts to emulate in a computer the mental modelling that occurs during human-to-human communication. Its focus is insuring the system serves the user in a cooperative manner. The user models in this research predict how a user will interpret an utterance the system is constructing.

There are significant differences between user modelling for critiquing and those that support advisory-dialog systems. The notion of a product constructed through a collaborative effort between the system and the user is central to most critics. Advisory systems are designed as *all-knowing* experts which, once they infer sufficient information about the user, will select or generate proper advice. The user's role is passive while in critics both the system and the user are active in solving the problem at hand. Advisory systems control the human-computer interaction. In critics, the system and the user share responsibility for solving the problem at hand and for guiding the interaction. There are several techniques developed in this area that can be integrated into a framework for models that support cooperative problem solving; they include: stereotyping approaches, the distinction between explicit and implicit acquisition techniques, and inference rules that use the content of the human computer dialog to enrich the user model contents.

2.2 User Models to Support Cooperative Problem Solving

There are three issues that need to be addressed for user modelling in cooperative problem solving system: how to represent the user model, how to acquire it, and how to access it. The first two areas are more difficult because access to the models is primarily determined by the representation. The acquisition problem, viewed in the ITS literature as a problem of diagnosis, is the most challenging. A topology was developed that summarizes the research literature by categorizing specific ideas and projects into the areas of: the knowledge the user model represents, how it is acquired, and its primary purpose.

It is useful to categorize acquisition techniques based on the directedness of the inferencing method.

1. **Direct acquisition techniques** are those where a specific piece of information is obtained by *explicitly* questioning users or from *implicit* observations of them. Usually a single characteristic about a user is inferred.
2. **Indirect acquisition techniques** are shortcuts, such as stereotypes or classification schemes; they are always implicit.

In the literature, the more commonly used distinction for acquisition techniques, (described best in [Kass, Finin 87]), is implicit versus explicit acquisition approaches, the orthogonality of these two categorizations is shown in Table 2-1.

Categorizing User Model Acquisition Techniques		
	Direct Techniques	Indirect Techniques
Implicit Acquisition	X	X
Explicit Acquisition	X	

Table 2-1: Two Orthogonal Classifications of Acquisition Techniques

The user characteristics represented in the model make a claim about what users can do; what they know; and their goals, plans, prejudices or preferences. To support the first two types of information, the representation must be in terms of domain expertise. Users do not simply know or not know a skill or domain entity, so it is inadequate to represent their knowledge using a binary value. Some student models tackle this problem by attempting to rate the knowledge of each domain entity in the user model using a linear value. Prevailing approaches that use ad hoc methods to set these values are often research efforts that orient on demonstrating how the acquisition process works rather than evaluating the validity of the models themselves. A simple approach is to represent each user according to a classification of domain expertise (e.g, expert, novice, beginner). For this research an alternative method for the system to categorize how well a user knows some piece of domain knowledge was needed.

2.2.1 Classifying the Users' Domain Knowledge

In [Fischer 88a] a useful schema for classifying users' knowledge , shown graphically in Figure 2-1, was presented. This schema provides provides a conceptual model for the space of user knowledge in the application domain. In general, the domains in the figure represent the following:

D_1: The subset of concepts (and their associated commands) that users know and use without any difficulty.

D_2: The subset of concepts which are used only occasionally, users do not know the details about them and may be unsure of their effects.

D_3: The user's mental model [Norman 82; Fischer 84] i.e., the set of domain concepts which are believed by an individual to exist.

D_4: This region represents the actual set of concepts in of a domain.

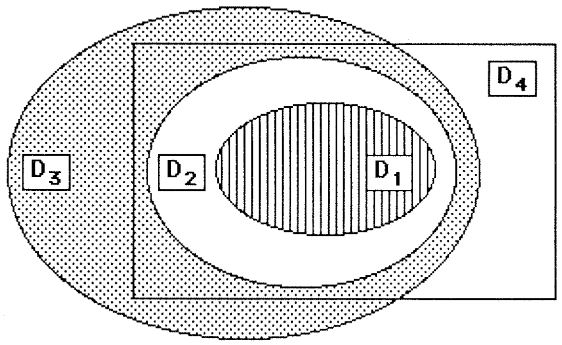


Figure 2-1: Levels of System Usage

Using this schema as a basis for the user model representation requires capturing how well a user understands domain entities according to these levels. The level at which users know a domain entity can, in turn, guide explanation giving.

2.2.2 Requirements for User Models in Cooperative Problem Solving Systems

Communication is at the heart of any cooperative effort. In order for a human and computer to collaborate effectively they must communicate about the product, and general information about the domain in the form of computer-produced explanations. Dialogs between the system and the user need to occur operate at a common level for which the system requires a model of the user. That user model needs to be accessible to other system components. Its contents will be used when presenting explanations, selecting items to analyze, and to record user preferences. The ultimate objective is an integrated system which adapts to users, allows them to specify their preferences, and is still consistent in the way it treats them.

The user models in cooperative problem solving systems have to be more individualized than classification schemes or stereotyping approaches. Users of a complex system are not homogeneous and the system needs to treat each one differently. Having individual models alone is inadequate, their contents have to change as the individuals knowledge improves — users most often become more knowledgeable or proficient over time. A precept of cooperative problem solving is to provide an environment that serves users not once but on a recurring basis; this means the system adapts and changes as users change. Achieving system adaptivity requires a representation of each user that is dynamic, persistent, and idiosyncratic .

Based on these requirements, several ideas in related research on user and student modelling were identified for incorporation into a user modelling framework for cooperative problem solving systems:

1. Stereotyping (GRUNDY)
2. Explicit and implicit acquisition methods (GUMS)
3. Representing user knowledge as a perturbation of the domain (Genetic Graphs)
4. Using the dialog content as a basis for acquisition inferencing (GUMAC)
5. Acquisition based on the relationship between knowledge as captured in the structure of the domain model (UMFE)

2.3 A User Model Architecture

The conceptual architecture for the user modelling component is designed to serve the needs of the critiquing paradigm but with an eye towards generality so that it might serve as the user modelling component for any cooperative problem solving system. The three major subcomponents of the architecture, the representation scheme, acquisition techniques, and access methods are shown in Figure 2-2.

2.3.1 Representation

The representation scheme is central because it must support acquisition and access. It must also be general, efficient, and easy to expand or modify. The two ideas from other research in user modelling that contribute to our representation scheme are: the use of a graph model for the domain, such as the genetic graph, and representing the user as an overlay of the domain model. The implementation we developed uses a more general approach than genetic graphs to represent the domain and a coloring of

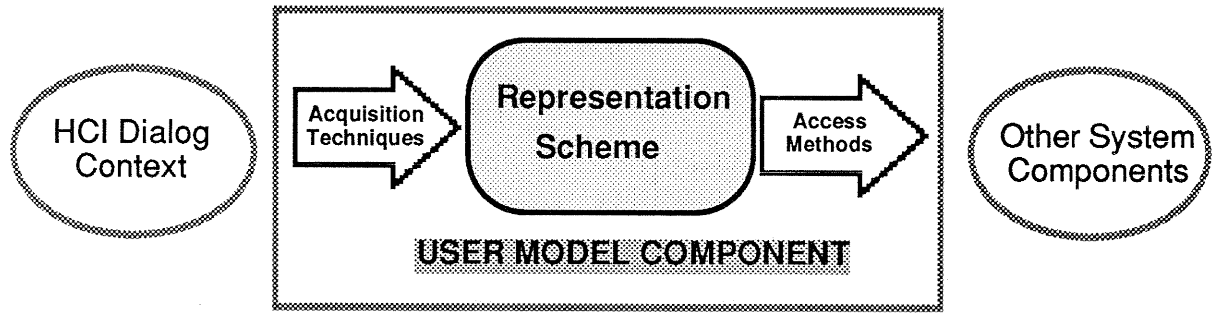


Figure 2-2: General Architecture for A User Modelling Component for CPSS

those graphs that is based on the schema for representing user domain knowledge.

2.3.2 Acquisition

Acquiring the user model is a complex function . It requires knowledge on the part of the system, knowledge about ways to infer the expertise level of the user. The acquisition methodology will need to support various approaches for acquiring information about a user.

The collection of acquisition techniques in the system can be conceptually viewed as a knowledge-based agent; an agent that can infer what a user knows from information provided by other system components which track the human-computer dialog; the agent also knows which questions to ask to infer the model contents, or select stereotypes, etc. Four categories of possible acquisition approaches were identified:

- *Explicit techniques* directly query the user for information that is entered into the user model. This is a suitable approach for obtaining an initial user model, it can be implemented as a simple questionnaire or testing session when a user accesses a system for the first time. It is not suitable during subsequent human-system interaction episodes because users are not willing to put up with such ancillary requirements more than once.
- *Implicit techniques* enrich the user model without interrupting the user. Two implicit techniques are of interest: stereotyping and the implicit implicature rules that operate on the human-computer dialog. Stereotypes are difficult to apply in many situations because, it is hard to determine what stereotypes to use. Some implicature rules can be modified so that they apply to the human-computer dialog present in most computer working environments. There are also implicit techniques that are indirect; they use the domain model structure to leverage information already known about a user.
- *Tutoring* methods acquire information from instructional episodes. These are episodes initiated either by user request or by the system for the express purpose of evaluating a user's knowledge. There are not yet any systems that attempt to do the latter but this appears to be a natural combination of ideas in tutoring and user model acquisition worthy of investigation. Given a comprehensive system, such as GRACE [Dews 89; Atwood et al. 90] that combines both a critic and a tutor, if a user voluntarily requests some tutoring, subjects addressed during tutoring can be used to enhance the student model.

- *Statistical* user model acquisition methods could be included in the implicit category but they are of sufficient interest to warrant a separate category. The technique is one of accumulating a history of user actions, and triggering inference methods upon reaching predefined threshold levels in that statistical history. In the ACTIVIST system models based on statistical methods proved to be effective [Fischer, Lemke, Schwab 84].

2.3.3 Access

The access methods are the third part of the architecture. The model contents must be accessible and usable by other components, or perhaps human agents. Access methods provide information to other system components about what the user does or does not know about the domain. The model developed in this research provides information to guide explanations, but the access methods are generic in nature so that they can also support what is needed for tutoring, advisors and so forth. Access functions need to be general enough to support known requirements, and flexible to accommodate extensions to the system. Access methods are not conceptually or theoretically difficult but are mostly determined by the *language* or methodology used to implement the representation.

2.4 Summary

A general framework for a user modelling component capable of supporting cooperative problem solving was developed; it incorporates techniques from research on student modelling in intelligent computer-aided instruction and user modelling in advisory dialog systems. An architectural framework for a user model was specified; it is able to support cooperative human-computer effort, is based on a categorization of users' expertise, and is general in nature. That conceptual framework has was instantiated, in part, in a user modelling component for LISP-CRITIC.

3. A Framework for Explanation

In building cooperative knowledge-based systems one objective is take advantage of the different strengths of users and computer systems. The system provides a source of expert domain knowledge that is used to make suggestions to users; the system should also explain those suggestions. Current explanation systems frequently fail to satisfy users for a variety of reasons; explanations are too often based on the implicit assumption that the process of explaining is a one-shot affair, and that the system will be able to produce or retrieve a complete and satisfying explanation provided it is endowed with *artificial intelligence*. Our approach takes advantage of available information and knowledge-based system technology to provide the user access to explanations at different levels of detail and complexity. Developmental efforts in this work focused on the concepts to be explained, rather than on selecting a complete prestored explanation appropriate for a given user. The domain and user models provide the system the capability to determine that set of concepts.

Early research on how to explain expert knowledge done in the context of MYCIN provided a rationale by showing the historical trace of the rules which fired [Shortliffe 76]. Rule-tracing explanatory approaches, even when "syntactically sugared" to make them more readable, are difficult to follow. Readers of that literature quickly realize that anyone not familiar with medical terminology and concepts have great difficulty understanding them. This points up a general shortcoming of most explanation approaches, they too often use domain concepts their readers do not know. User models help systems overcome this shortcoming.

3.1 Theory

The theoretical approach used in this research considers why knowledge-based cooperative systems need an explanation capability, the precise function the explanations provide, what is wrong with current approaches, and the basis for the approach we use.

3.1.1 The Need for Explanations

In order for professionals, managers, and scientists to accept knowledge-based systems, it is essential to provide explanations of the knowledge. The need for good explanations was identified in a study of physicians' attitudes towards expert systems:

> *Explanation.* The system should be able to justify its advice in terms that are understandable and persuasive. In addition, it is preferable that a system adapt its explanation to the needs and characteristics of the user (e.g., demonstrated or assumed level of background knowledge in the domain). A system that gives dogmatic advice is likely to be rejected. [Teach, Shortliffe 84, p. 651]

Explanation in cognitive science evokes two different meanings: the process of presenting information, and an internal cognitive process that develops a knowledge representation. Our work focused on the presentation process while attempting to keep both meanings in mind. The internal-process view claims that explanation is equivalent to understanding [Schank 86]. According to this perspective, humans achieve understanding by generating their own explanations. For this work the system must provide the information needed to support the self-explanation process. If systems know what information is required to insure understanding, can tailor that information to the individual, and then present it in an optimal form, users might adopt it as their own. But this is too ambitious, so rather than attempting complete, ideal explanations for each user to integrate into their mental models, computers must instead concentrate on providing users with the material required to produce their own explanations. This means *generating explanations with the computer* not merely displaying stored ones; explanations that use the domain concepts appropriate to a particular problem-solving context, provide users the opportunity to produce a self-explanation and achieve understanding.

3.1.2 Functions for Explanations

We investigated how to design systems that serve users actively engaged in their own work — cooperative problem solving systems that provide a task-based environment in which users work toward goal accomplishment. Systems that support users' work are more than media used for describing their problems, and more than tools to extract useful information from a database. They should be active agents that provide problem-domain communications [Fischer, Lemke 88], can critique users' work, and can explain their knowledge. We analyzed the reasons users seek explanations and determined that a common triggering condition is an impasse [Mastaglio, Reeves 90].

Assisting users during problem solving requires that explanations be designed to help them overcome impasses. Explanations in cooperative problem solving systems can serve four functions similar to the ones found during a study of the users of knowledge-based medical information systems [Wallis, Shortliffe 84]:

1. Explanations allow users to examine the system's recommendations.
2. Users need explanations to relate recommendations to domain concepts — to understand "what is suggested".
3. The explanation should help users to see the rationale for recommendations — to understand "why this would be better".

4. Explanations are needed by users to learn the underlying domain concepts.

These functions are not mutually exclusive; single explanations in a cooperative problem solving system will have to accommodate multiple purposes.

3.1.3 Shortcomings of Current Approaches

Most attempts to provide explanations use prestored scripts in the form of canned-text. Those types of descriptions have been criticized as difficult to understand, incomplete, and hard to navigate [Weiss 88]. Empirical studies of tutoring in both humans and computers determined that canned explanations are insufficient approaches [Fox 88]. Because critiquing and tutoring are closely related, many of the problems listed there apply to critiquing as well. The use of canned explanations is not inherently bad just because it is done by a computer. Empirical studies of human explanations found people often employ a similar strategy when explaining something "for the sake of others" [Schank 86]. The difference is that people understand their prestored explanations — they make sense to the explainer, and they represent a form of mental model. When it happens that the recipient does not understanding such an explanation, human-to-human discourse allows them to query the explainer for clarification or elaboration.

Some systems attempt to overcome several of the problems of canned text, but none addresses all shortcomings. Our strategy is to recognize the shortcomings while using an interactive approach based on available technology integrated with domain and user modelling capabilities. We consider the limitations of canned-text but try to be realistic about current capabilities of computer systems. In many theoretical explanation strategies there is an assumed environment containing an intelligent computer able to generate natural language, predict users' needs, and enter into a followup dialog. Techniques are needed that work within the constraints of available technology. Based on these limitations, a minimal explanation framework was developed.

3.1.4 Basis for Minimalist Explanations

If a user model can provide a detailed representation of users' knowledge, then it will be possible to formulate and present an appropriate explanation. One method to achieve that is the minimalist approach [Fischer, Mastaglio, Reeves, Rieman 90]. The ideas for minimal-explanations share underlying theoretical foundations with minimal approaches to instruction [Carroll, Carrithers 84]. Theoretical bases for this approach are found in work on discourse comprehension:

1. Short-term memory is a fundamental limiting factor in reading and understanding text [Dijk, Kintsch 83; Britton, Black 85]. The best explanations are those that contain no more information than absolutely necessary, since extra words increase the chances that essential facts will be lost from memory before the entire explanation is processed.
2. It is important to relate written text to the readers' existing knowledge [Kintsch 89; Fischer et al. 88].

Similar practical guidelines are also found in the theory of rhetoric. Flesch developed formulae to evaluate the readability of text [Flesch 49] which are frequently used to evaluate documentation and instruction. Computer explanation systems should comply with similar standards; using short sentences and known vocabulary are important criteria. Strunk and White's guide to good writing contains similar advice; they tell writers "Don't explain too much" when writing explanatory text [Strunk, White 57].

3.2 Related Work

Some research on explanations in knowledge-based systems assumes a natural language interaction, for example dialog advisory systems. Another approach attempts to capture expertise during the knowledge acquisition phase of building an expert system; using a methodology which will later facilitate explaining that knowledge: the explainable expert system approach (EES) developed by Swartout [Swartout 83] is one example.

Moore extended the EES work, in a program-transformation system similar to LISP-CRITIC [Neches, Swartout, Moore 85]. Her research addressed the situation where users need to follow-up on explanations for clarification. Her "reactive" approach provides the user with an initial explanation, but accommodates the situation where it fails to satisfy the user; it provides increasingly informative fall-back explanations [Moore 87]. Her framework achieves a fall-back capability by monitoring and recording the dialog between the system and users, then using this dialog trace to identify and overcome difficulties. Moore agrees that a convivial system should make a good-faith effort to provide the right explanation the first time [Moore 89]; it is when that fails that her reactive approach or something similar is needed. Providing the best possible initial explanation requires the system to understand the domain at a level that supports the generation process and the modelling of its users [Kass 87b].

Natural language approaches, such as Kass's reliance on Grice's rules for cooperative conversation and Moore's iterative fall-backs, use human-to-human discourse as their model for human-computer communication. This may be unreasonable, especially given the difficulties of reading large sections of text from a computer screen [Hansen, Hass 88]. Knowledge-based system designers need to recognize the special capabilities and limitations of computers rather than trying to coerce the natural language paradigm into a screen- and keyboard-interaction style [Kennedy etal. 88; Fischer 88b].

Paris [Paris 87; Paris 89] developed an approach to explanation based on an assumed user model. Her work provided initial motivation for our user modelling investigations [Mastaglio 90b]. Her theory builds hybrid textual descriptions for devices using two strategies, a process trace and a constituency scheme. A hybrid explanation for a device is actually a mixture of the two methods based on what users already know. Those *constituents* with which a user is familiar need only be described as component parts of the device being explained, but others need to be described in terms of how they operate (their process), or their own constituents, and so on. The process executes recursively, capturing those portions of the domain (objects or concepts) that an individual user needs explained.

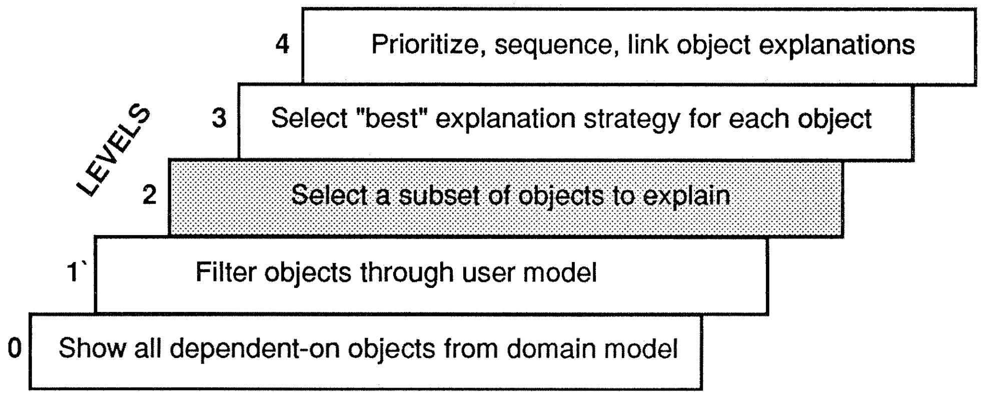
Requiring a knowledge-based system to have a concept level domain model was also a finding of research by Chandrasekaran and associates. They investigated the need for deep domain models in expert systems [Chandrasekaran, Tanner, Josephson 88; Chandrasekaran, Tanner, Josephson 89]. Their theoretical framework claims that explanation involves three issues: presentation, modelling the user, and endowing a system with "self-understanding". They propose a "generic task methodology" approach to building the expert system focused at the task level rather than abstract domain concepts. It makes basic explanation constructs available at a level of abstraction closer to the user's conceptual level and therefore similar to the work on human problem-domain communications [Fischer, Lemke 88]. It also uses general domain knowledge to justify the system's problem-solving approaches.

3.3 An Explanation Framework to Support Critiquing

To support explanation in critics requires sufficient knowledge on the part of the system to describe what is going on and why.

A domain model provides a conceptual basis for an explanation in terms of those functions and concepts that are prerequisite knowledge. Determining prerequisite knowledge is a recursive process because understanding those domain concepts that are prerequisites for the given concept requires, in turn, understanding their prerequisites and so on. To support such an approach, the deep structure in the domain model is queried to obtain a *concept-set* comprised of those prerequisites. A satisfactory explanation approach still needs to do something more, it must identify the concepts in that set that do not require explaining because the user already knows them. Furthermore it will reason about the best way to explain the remaining concepts.

We investigated ways to organize explanations for a system such as LISP-CRITIC and developed a framework that includes different levels for explanations (shown in Figure 3-1). The explanation levels capture necessary and sufficient conditions for adequate explanations. Each level incrementally enhances work done at a lower level, integrating additional knowledge about the user and the domain. A Level 0 explanation does not require knowledge about individual users. It uses the domain model to meet a necessary condition — knowing what to explain. The explanation component is provided the set of prerequisite concepts required to understand an object needing to be explained. Level 1 brings the user model into the process; here the prerequisite set of concepts is "filtered" through the user model to determine the subset appropriate for a given individual. In many cases, that filtered set is still probably larger than we want to explain in a single episode. Therefore, at Level 2 the explanation component needs to know strategies that determine exactly which of that subset to explain and how.


4
Prioritize, sequence, link object explanations
3
Select "best" explanation strategy for each object
LEVELS
2
Select a subset of objects to explain
1`
Filter objects through user model
0
Show all dependent-on objects from domain model

Five levels of explanation are identified. Level 0 insures all prerequisite knowledge for a given domain object is available to the explanation component. Level 1 builds on level 0 and so forth. The current LISP-CRITIC system provides simplified level 2 explanations. Level 3 and 4 require presentation and natural language generation techniques.

Figure 3-1: Explanation Levels

A system operating at Level 2 passes a sufficiency test: it knows *what* to explain to an *individual* user in a specific situation. However, it is still faced with the presentation problem; explanations need to be presented in a manner and style that will make them more readable — the system has to know *how* to do the explaining. Achieving this level means the system will need to make use of additional domain knowledge or other information in the user model in order to determine a "best" strategy for explaining a concept.

3.4 Role of the User Model in Explanations

The user model is discussed in detail later but, because its purpose is to support explanation generation, it is important to consider especially the criteria for the user model that are established by the explanation framework.

Cooperative systems must tailor their explanations to individuals, the basis for tailoring is the role of the user model. A simple approach is to classify users by their expertise (e.g., novice, intermediate, expert); but this is not a valid representation for many domains and users. A finer grained representation that follows from Paris's work, represents user's knowledge in terms of the domain objects and concepts.

The user model needs a representation of the user's domain knowledge detailed enough to support the five "levels" of explanation shown in Figure 3-1. It has to be based, at least in part, on the conceptual model of the domain, so that it can filter the set of concepts that form the explanation basis. The model needs to capture the user's goals in order to support Level 3 explanations. Supporting Level 4 explanations is more difficult and beyond the state of current research. We will not know all requirements for user models to support this level until that research matures.

The user model that was developed provides support for explanation giving according to the framework presented. One supported approach is the minimal-explanation strategy; it interrogates the user model to determine a minimal set of concepts to explain. Such strategies are possible because the model knows which concepts are familiar to the user.

4. LISP-CRITIC System Overview and LISP Domain Model

LISP-CRITIC was the development environment used to implement the user modelling framework. It is a knowledge-based system that is designed to support programmers in the context of their work. It does not have "automatic programming" capabilities but operates according to same principle of "intelligent assistance" that is fundamental in the PROGRAMMER'S APPRENTICE work [Rich, Waters 90].[1] In the terms of that research LISP-CRITIC belongs to the class of what are called "transformation systems" [Rich, Waters 88].

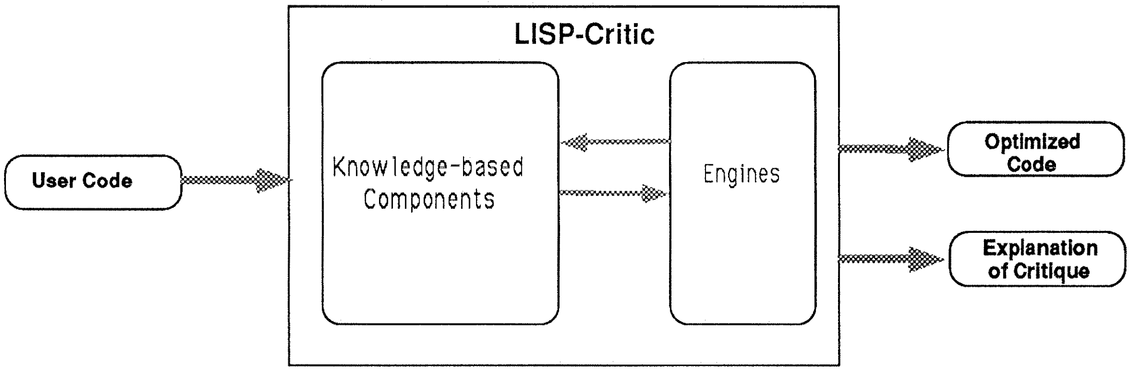
Comparisons between LISP-CRITIC and LISP Tutor are inevitable but the purposes for the two systems are quite diverse. LISP Tutor teaches programming by leading students through exercises known to the system. LISP-CRITIC is oriented toward aiding programmers involved in real work by suggesting better ways to implement a specific program. In LISP-CRITIC learning will inevitably occur but it is best to design

[1]The long term vision for the PROGRAMMER'S APPRENTICE is that it "act as a software engineer's junior partner and **critic** (emphasis added)" [Rich, Waters 90, p. 1]. In our view, development of LISP-CRITIC provides significant understanding of what is involved in the critic portion of such a system.

the system to make that learning as effective as possible.

4.1 Description

LISP-CRITIC allows interaction between the system and the user at the level of individual transformation; it provides context-specific tailored explanations upon request, and supports some user adaptability. Instead of transforming an entire LISP program, handing it back to the user, and trying to explain the differences, the design is based on an assumption that to achieve a more collaborative style, users should be able to decide, on a transformation-by-transformation basis, whether or not to make a change. Furthermore the system has to be able to change the code the user actually wants modified, while leaving the rest of the program intact; the resulting program must still compile and execute properly. Users need explanations of any single suggestion. These goals led us to the develop a version that enhances an existing, commonly used, program development environment, the Symbolics ZMACS editor; users can access the critic at any time while they are editing LISP code in ZMACS (see Figure 6-2). The critic examines the code and makes one suggestion at a time; the programmer can accept the recommendation, reject it, or request an explanation. When a transformation is accepted the system changes the code in the editing buffer.



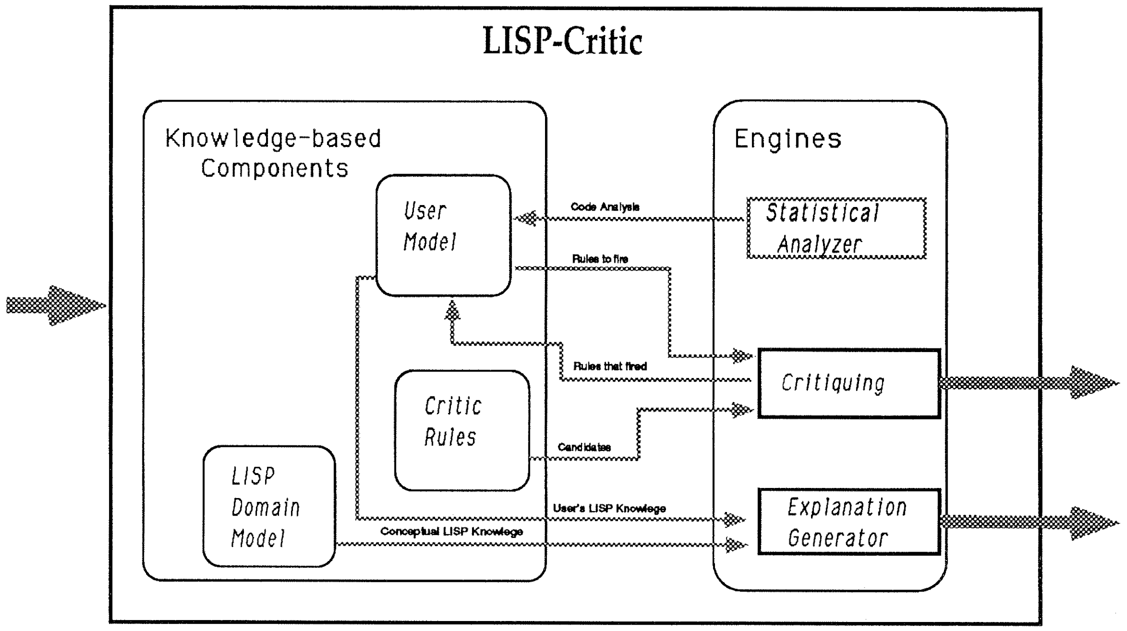
This figure shows the architectural components of LISP-CRITIC and the general flow of data.

Figure 4-1: The Architecture of LISP-CRITIC

A general overview of the system architecture is shown in Figure 4-1. The user's code is analyzed at the s-expression level. When an opportunity is found to improve that expression, the systems provides an improved (optimized) version and, if the user requests it, an explanation. Inside of LISP-CRITIC are a set of engines and a set of knowledge-based components. Figure 4-2 shows the internal components and the flow of information between them.

4.2 The LISP Domain Model in LISP-CRITIC

The domain model developed to support explanation-giving and user modelling in LISP-CRITIC refines and implements ideas developed in previous research [Fischer 88a]. The domain model is an enabling technology required for both the explanation and user modelling components. The model represents the domain of LISP in terms of three entities: the concepts of the language, basic COMMON LISP



This figure shows the internal components of LISP-CRITIC and the information flow between them.

Figure 4-2: Internal Components of LISP-CRITIC

functions, and the transformation rules in LISP-CRITIC.

4.2.1 Requirements for a Domain Model

In order to achieve cooperative problem solving systems must have the capability to explain their reasoning. In the case of critiquing, in general, and LISP-CRITIC in particular, this means explaining the reason for certain suggestions — the rationale and concepts behind a rule.

A common theme in other research is that in order to provide an acceptable explanation capability, the system needs to represent knowledge of the subject domain at an abstract level [Clancey 87; Kass, Finin 88b; Paris 87; Wiener 80]. For LISP-CRITIC, previous investigations determined the possibility of representing programming knowledge in terms of concepts, programmer goals and functions [Fischer 88a]. Such a representation can explain the improvements suggested by the rules and derive a model of the user.

The rule base in LISP-CRITIC represents procedural knowledge in a compiled or applicable form that is appropriate for efficiently analyzing code and rapidly generating recommendations for how to improve that code. However, knowledge in this form will only support rule tracing explanation approaches [Scott, Clancey, Davis, Shortliffe 84] and these were shown to be inadequate [Clancey 84]; systems need the ability to explain at the concept level to facilitate user understanding and support learning. This requires a more abstract domain model; a model that captures the abstractions representing the domain conceptual level.

A conceptual structuring of the domain links the applicative rule-based knowledge with explanation strategies and with the user model. In LISP-CRITIC, a rule, or set of rules, is the underlying causative mechanism behind a single piece of advice. To understand that advice well enough to decide whether or not to accept it, users needs to know the concepts behind that rule. A concept-based domain representation provides that information. In turn, the system determines which concepts are not part of the user's knowledge so it can focus on explaining the unfamiliar ones.

4.2.2 Form of the Lisp Domain Model

In order to support the explanation strategies and user modelling process in Lisp-Critic, the information contained in the domain model consists first of the underlying concepts for Lisp. To determine those concepts we reviewed the following commonly used Lisp texts: [Steele 84], [Winston, Horn 81], and [Wilensky 84]. Forty-five commonly-referred-to concepts were identified in these texts. The list does not include more fundamental concepts that exist "in the world", such as the set of integers, but focuses on those concepts that are unique to Lisp or programming languages in general. The terms used to identify these concepts are listed alphabetically in Figure 4-3. The term concept, as expressed in the Philosophy of Science literature, is an abstract notion; there is a distinction between concepts themselves and the terms that stand for them [Hempel 65]. The concepts shown in Figure 4-3 were designated using terms that seemed appropriate, while recognizing that in other research, and context, they may be described with other names.

ARGUMENTS
ASSOCIATION-LISTS
CAR-CDR-CONCATENATION
CONDITIONAL-EXITS
CONDITIONALS
CONS-CELL
DATA-TYPES
DESTRUCTIVE-FUNCTIONS
DOTTED-PAIR
EMBEDDED-FUNCTIONS
EVALUATION
EVALUATION-ORDER
FALSE/EMPTY-LIST/NIL
FUNCTION-DEFINITION
FUNCTIONS
IDENTITY-VS-EQUIVALENCE
INPUT-FUNCTIONS
INTERNAL-REPRESENTATION
ITERATION
LAMBDA-BINDING
LISP-ATOM
LIST-ITERATION
LISTS
LITERAL/QUOTE
LOGICAL-FUNCTIONS
MAPPING
MULTI-VALUE-RETURN
NUMERIC-ITERATION
OPTIONAL-PARAMETERS
OUTPUT-FUNCTIONS
PARALLEL/SEQUENTIAL-BINDING
PARAMETERS
PREDICATES
PROPERTY-LISTS
RECURSION
SCOPE
SIDE-EFFECTS
STRINGS
SYMBOLIC-EXPRESSION
TAIL
TESTS
TRUE/NON-NIL
VARIABLE-INITIALIZATION
VARIABLES

Figure 4-3: List of Domain Concepts

While selecting the set of concepts for inclusion in the domain model, two types of relationships between concepts were identified, relationships that are useful for explanation giving, and ones which can be used in user model acquisition:

1. **The dependent-on relationship:** This indicates for a particular concept which other, more fundamental, concepts are prerequisites to understanding it.
2. **The related-concepts relationship:** This is a relationship between concepts that are similar, this information could be used by the explanation component in some presentation strategies.

Also 103 fundamental LISP functions are represented in the domain model, primarily those that are found in the LISP-CRITIC rules. The term "function" is not entirely correct, this class of domain entity might more specifically be referred to using the term "constructs", as in [Steele 84].[2] To understand a function also depends on understanding certain underlying concepts; therefore, functions are related to concepts via "dependent-on" relationships, similar to the one described above. Functions may also be similar to one another, for example, *cond* is similar to *if*: and the model also captures this relationship.

We also capture LISP-CRITIC rules in the domain structure because this is the level of application knowledge needed for a complete single representation of the system knowledge. A rule is linked to the functions that occur in the rule, and the LISP concepts that underlies it. When the system recommends a change to a program, the only thing it knows is that some code conformed to a pattern expressed on the left hand side of that rule and that it could be rewritten according to the pattern on the right hand side. To model what is involved in understanding that rule, it was necessary to capture knowledge about the functions in the rule and any domain concepts that are behind it. Concepts and functions probably exist as part of a programmer's mental model of the domain [Gentner, Stevens 83], therefore, these parts of the domain model may be something close to a cognitive representation, possibly representing chunks. It is unlikely that users, with a few exceptions, retain a LISP-CRITIC rule as part of their mental model for the domain, even after they develop an understanding of it.

In summary, the domain model captures three types of entities LISP concepts, LISP functions, and the LISP-CRITIC rules, together with the relationships between instances of them. Relationships are often one-to-many, but their topology, although somewhat hierarchical within certain relationships (like the dependent-on-concepts for all concepts in the model), is highly interconnected and acyclic. Several paradigms, such as, frames and semantic networks were considered as possible representation schemes for the model.

4.2.3 Implementation of the Domain Model

In the domain model implementation, the concepts were defined as classes, and relations between concepts, captured in slot definitions. The class hierarchy for LISP consists of a super class, *lisp-object*, with three subclasses *lisp-concepts*, *lisp-functions* and *lisp-critic-rules*. There are slots in each object instance for name; dependent-on-concepts, and, as appropriate, related-concepts, related-functions, and related-rules. The CLOS code that defines these objects is shown in Figure 4-4. The three types of entities inherent common slots for *name* and *dependent-on-concepts* from the fundamental class *lisp-object*.

The complete domain model is difficult to show graphically because it is highly interconnected. It can be considered to have three layers, one each for LISP concepts, functions, and LISP-CRITIC rules. Populating each layer are instances of the entity class for that level. Links are found between instances

[2]Still more precisely, the set actually consists of special forms and standard macros defined for COMMON LISP.

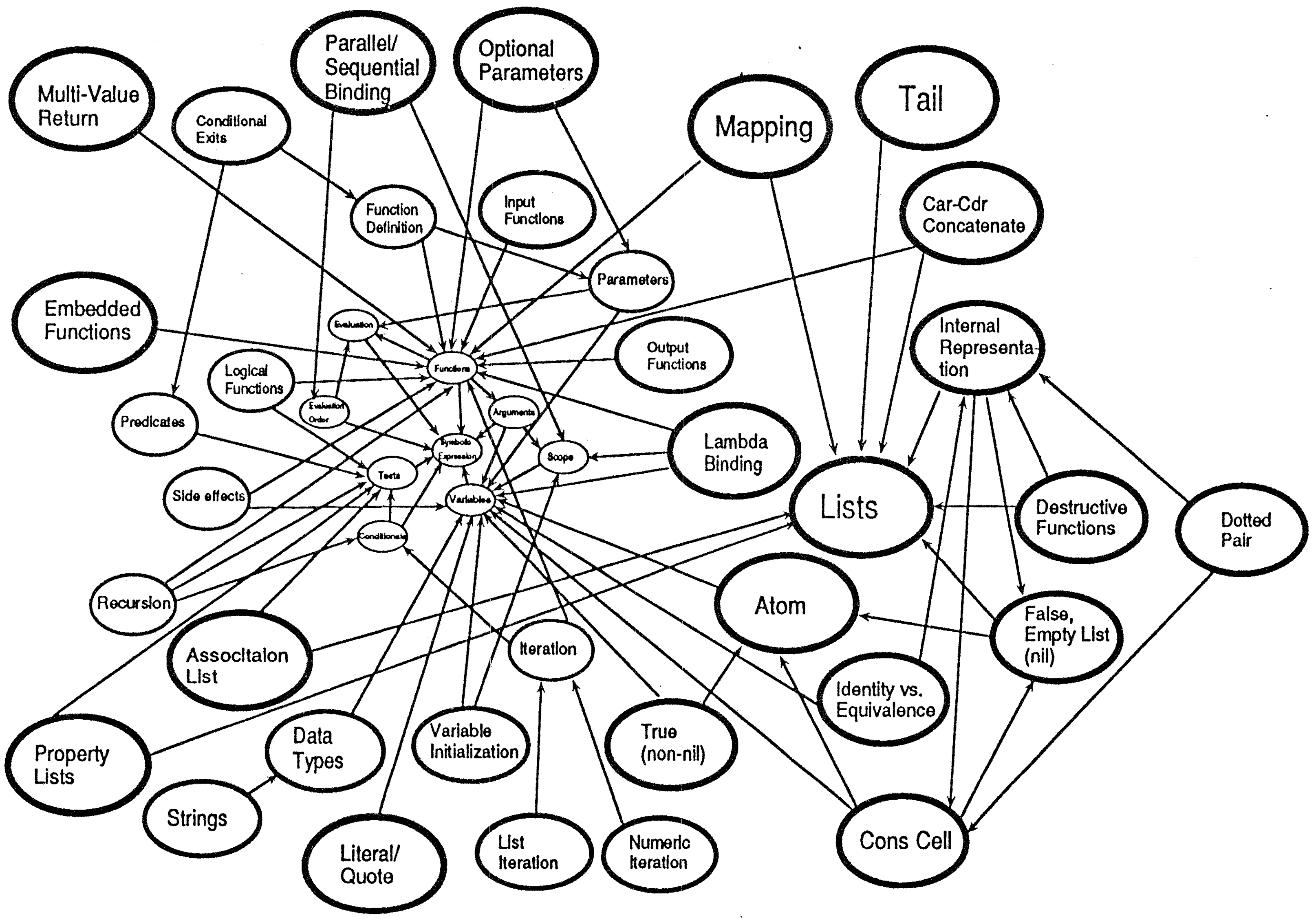
```
(defclass LISP-OBJECT()
     ;;; Generic Super Class for all LISP Objects
     ((name
     :accessor name
     :initarg :name)
      (dependent-on-concepts
        :initform  nil
        :accessor concepts-dependent-on
        :initarg  :dependent-on-concepts)))

(defclass LISP-CONCEPT (lisp-object)
     ((related-concepts
        :accessor related-concepts
        :initform nil
        :initarg :related-concepts)
      (level
        :accessor level
        :initform 'high-level
        :initarg :level)))

(defclass LISP-FUNCTION (lisp-object)
     ((pattern
        :accessor syntax-pattern
        :initarg :pattern)
      (related-functions
        :accessor related-functions
        :initform nil
        :initarg :related-functions)
      (category
        :accessor category
        :initform 'unclassified
        :initarg :category)))

(defclass LISP-CRITIC-RULE (lisp-object)
     ((functions-in-rule
        :accessor functions
        :initarg :functions-in-rule)
      (related-rules
        :accessor related-rules
        :initform nil
        :initarg :related-rules
        )))
```

Figure 4-4: CLOS Specification For LISP Domain Entities

within a level, as well as between instances at different levels. For example the LISP-CRITIC rule *cond-to-if* is found in the LISP-CRITIC rule layer; it is linked both to similar rules (e.g. *cond-to-when*) within that layer, as well as to concepts (e.g., *conditionals*) in the concept layer, and of course to functions (e.g., *cond*) in the function layer.

To give the reader a flavor for the interconnectivity of the model, again consider the LISP concept *recursion*. Understanding *recursion* is dependent on the user understanding the concepts of *tests, conditionals*, and *functions*. *Recursion* is also related to the concept of *iterations*. The code to instantiate *Recursion* as an instance of a *lisp-concept* is shown in Figure 4-6. Most concepts, functions, and rules in the domain model have similar high degrees of connectivity.

It is difficult to display the entire domain structure in a single two dimensional graph. Figure 4-5 provides a feel for the complexity of the structure when viewed across a portion of a single strata or level of the domain. It shows graphically the *lisp-concept* layer together with the *dependent-on-concepts* links between the 45 concepts. For simplicity and readability sake, each oval represents a LISP concept, and links are all of the same type; they represent the dependent-on-concept relation.



This graph shows the dependent-on-concepts relationships for the LISP concepts in the Domain Model

Figure 4-5: Concept Layer of Domain Model

```
(make-instance 'lisp-concept
                    :name 'recursion
                    :dependent-on-concepts
                         '(tests conditionals functions)
                    :related-concepts '(iteration)
                    :level 'intermediate)
```

Figure 4-6: CLOS Specification For Concept Recursion.

The explanation component uses the domain model to determine what concepts must be explained to a user, Recommendations are generated from a rule firing therefore the user needs to understand the concepts a rule depends on, as well as the functions that are part of that rule. The system must explain to the user those concepts and functions the user does not already know. Furthermore, if the user does not understand the more fundamental concepts upon which the understanding of a given concept is dependent, the system may want to explain those as well. For the concept, *recursion* the user must already know the concepts: *tests, conditionals and functions* or these must be described as part of the strategy for explaining *recursion.* The explanation system could also use the domain model to select an explanation strategy by using the related-concepts or related-functions relationships (slots). In the case of *recursion*, the domain model indicates that *iteration* is a related concept so one strategy would be for the system to compare recursion to iteration.

The user modelling component uses the domain model for two purposes. The model provides a representational basis for users' knowledge; the user model overlays the domain model to capture the LISP concepts and functions that a user already understands. It is an annotation or coloring of the graph representing LISP. The user modelling component has a set of inference methods, described later, which build up individual models representing each user.

5. The User Modelling & Explanation Components in LISP-CRITIC

Two major system components: the user modelling component and the explanation system were added to LISP-CRITIC as a part of this project.

5.1 User Modelling Component

The user modelling component acquires the user model which represents the knowledge of each user in an object-oriented structure. This component was implemented in the Common LISP Objects Systems (CLOS). Access to individual models is provided via a set of generic interface functions; other system components know which functions to call to obtain required information from the user model. The user modelling component invokes the appropriate methods to actually access a user model's contents; it uses the domain model structure to insure that appropriate information is provided. The acquisition subcomponent contains direct methods that are based on episodes from the user-computer dialog, and indirect methods that trigger changes to individual models.

5.1.1 Design Approach

The design objectives for the user modelling component are based on the goal of supporting the explanation-giving framework. The specific implementation approaches selected to achieve these objectives resulted from the analysis of other user modelling research, plus the requirements and framework for user modelling needed to support cooperative problem solving.

The user modelling component design provides support for explanation-giving, accommodates various acquisition techniques, and is able to represent a variety of information about the user. The model captures users' domain knowledge and supports implicit updating. An object-oriented approach was selected for implementation in order to insure that the model is extensible, can be adapted to accommodate other techniques and can be easily modified. The object-oriented approach allows new methods to be defined on existing slots in the model and new slots to be added to the model's class definition. The component supports both implicit and explicit update. In this research the implicit update techniques were the primary focus however, the functions which modify the content of the user model are general methods designed to support other acquisition approaches as well. Methods that save the contents of the model at the termination of a user-system dialog and start the next dialog with that model are included in the component.

The model supports changes in users' knowledge over time. It is in this sense that the model is dynamic; its contents change as a user's knowledge improves. The emphasis in developing inference methods was on improvements in users' knowledge of the domain. That the model is at best an approximation of the user implies that the modelling component must be designed to include techniques for improving that approximation. Whatever information is available to enhance the model has to be used to best advantage.

Three different approaches to representing users of were considered classification methods, stereotyping, and an overlay of the systems domain knowledge (the LISP-CRITIC rules). Initial attempts to model the user with classification methods in support of explanation-giving [Frank, Lynn, Mastaglio 87] met with only limited success. The problem with classification methods were two-fold. First, canned-text explanations directed at a particular level of expertise were found to be unsatisfactory; often they were were too basic to satisfy the user, or difficult to understand because they used concepts not yet understood. Another problem is how to classify a user into one of the prespecified categories; the ones used (novice, intermediate, and expert) did not capture individual expertise in a satisfactory manner. The second problem with classification methods is that they are not fine grained enough to provide adequate fidelity in their representation of individuals.

A schema for model acquisition using stereotypes of LISP programmers was developed [Fischer, Mastaglio, Rieman 89]. It was based on Rich's approach to stereotyping [Rich 79], and showed promise as a way to leverage analysis of the content of users programs to *stereotype* them, and from that stereotype infer additional characteristics. A study in which human LISP experts were provided a program and asked to assess the expertise of the programmer showed that the human experts either looked for or noticed what we called "cues" in the code; cues trigger inferences about the expertise level of the programmers who wrote the code. Methods based on this idea were developed and but this line of research confronted the problems of which stereotypes to use, where they come from, and how to validate them.

Representing a user's knowledge as an overlay of the existing rule base was also considered. It was found to be useful for guiding critiquing (e.g., making it more efficient by enabling or disabling rules). However, a model that only captures user knowledge in terms of the transformation rules is inadequate; it cannot provide the required support for explanation-giving. Slots in the model represent rule level information for a user and provide an overlay of the rule base.

The limitations encountered in these other approaches provided a key objective for the design of the user modelling component, to implement a model that represents user knowledge of the domain at a level that is of fine enough granularity to support the explanation of domain entities. The basis for that representation turned out to be the same type of domain model required to accomplish explanation-giving.

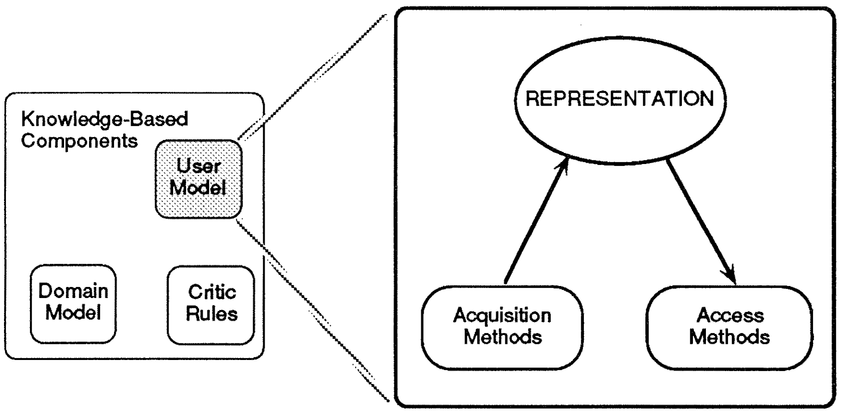
An object-oriented representation allows the model representing each individual to be idiosyncratic but for all the individual models to confirm to a common format. This requirement, coupled with the need to support easy access and the changing of separate instances of the entire class of models, led to an object-oriented representation. The structure of the individual models is defined as a class, and communicating with the instances of that class is facilitated through methods defined on it. The object-oriented representation can also support potentials enhancements to the overall user modelling component. The need to achieve a representation of users' domain knowledge in a more abstract or conceptual form than rules resulted in basing the user model on the conceptual domain model.

To support dynamic update without explicitly querying the user, the implementation uses information available in the context of the human-computer dialog. Dialog, in the sense used here, refers to any action that occurs between the system and the user. The idea that the model should be implicitly enhanced based on the dialog led to an analysis of the content of these interactions. The dialogs consist of a series of episodes in which one of the following occur:

- user requests and receives an explanation of a critic suggestion
- user decides to accept (or reject) a critic suggestion
- user accesses additional on-line documentation to help clarify an explanation
- user informs LISP-CRITIC to disable (enable) a rule
- user adds a personal comment to an argumentation database about the applicability or usefulness of a transformation in the rule base

The implementation uses what takes place in the episodes to trigger system inferences about the user. A basis for this approach is the fact that users apply their knowledge when constructing their "side" of the dialog, therefore any actions they take provide evidence about what they know. Just as significant is that when users are receivers of information there are cues here about how the user's knowledge should be changing. Specifically, they should now have command of the domain concepts explained by the system. In this second case, users "learn" from what the system tells them — this is the basis for some of the direct inference methods that will be described later.

The above objectives and approach guided the manner in which the user model is represented, acquired, and accessed. An architectural overview of the user model component in Figure 5-1 shows the separate subcomponents and functions of LISP-CRITIC's user modelling component; it corresponds to the general architecture developed and shown in Figure 2-2 and is an internal view of the user-model component of the overall system diagram that was shown in Figures 4-1 and 4-2.



The user modelling component is one of the knowledge-based components of LISP-CRITIC. Data are indicated with an oval, collections of processes with rectangles, data flow with directed arrows. The three subcomponents are: a **representation** in object-oriented form (CLOS), **acquisition methods**, and **access methods**. Acquisition methods modify the representation — information flows from them to the representation. Access methods extracts information from the model — information flows from the representation.

Figure 5-1: User Model Component for LISP-CRITIC

5.1.2 User Model Representation

The representation is designed to capture a variety of information about the user. The interesting part of the model (with respect to this project) are those slots that represent the user's expertise in the domain of LISP: *rules-known*, *functions-known*, and *concepts-known* slots. Conceptually these record the coloring of the domain model graph for the user. An approach considered for the representation was an overlay of the domain model; the overlay representing those domain entities a user knows. But the model needs to also capture the levels of the users' knowledge according to the classification framework, the approach is to model the user as a coloring of the graph representation of the domain.

According to the conceptual framework, a user's knowledge about a given concept can be categorized into one of three levels: d1, d2, and d3; according to the framework shown in Figure 2-1; it provides a useful scheme for approximating the expertise levels of users. For this research, we adapted it to represent user knowledge of a programming language. For LISP the regions in the graph are interpreted as follows:

D_1: The subset of LISP functions and underlying language concepts which users know and incorporates into their programs regularly, they understand these quite well.

D_2: The subset of concepts which users know and will use, but only occasionally. They does not know the details nor perhaps even the specific syntax of functions in this region but are aware of their existence and have a general understanding of their purpose. Users might refer to a LISP text, on-line documentation, or consult a colleague for help in coding functions in this class. Concepts in this class are less well understood by users than those in D_1 but still can be considered a part of their active knowledge.

D_3: The conceptual model of LISP held by a user. The concepts and functions that they think exist in the language; this region also includes misconceptions.

D_4: The domain knowledge of LISP.

Inference methods recognize domain knowledge in d1 and d2. The methods conceptually marks entities in the domain as: well known to the user (d1), known to the user but not well (d2), and unknown (d0). Entities at level d0 are not explicitly listed in the user model but the system infers that lack of knowledge by their absence from the appropriate slot. It may also be the case that the system has not yet encountered anything to trigger an inference about the level of knowledge — it just does not know how well the user knows the entity, if at all.

Each individual's model is an instance of the class user model. A user's knowledge about each category of domain object is captured in slots in that model. For example one slot contains the LISP functions a user knows as well as their levels. Other slots contain personal information and data about the user's background. The information and data slots can be filled during initial start-up of LISP-CRITIC for a particular user (i.e., the first time it is ever invoked by them) using explicit acquisition techniques.

5.1.3 User Model Acquisition

The user modelling component contains a collection of methods that infer which domain concepts belong in the user's model, and the level of that knowledge. The information that triggers these methods is passed to the user model by other system components using interface functions; in turn, indirect methods get internally triggered by those changes.

This research investigated how a user model for a cooperative problem solving system could be enhanced incrementally over multiple dialog episodes using implicit acquisition. To review the point made in Table 2-1, one finding was that the system can use two different classes of update methods:

- Direct methods that use a specific dialog item to trigger singular changes in the user model information. They make changes to the user model as a *direct* result of information the user receives from the system, or of an action the user takes.
- Indirect methods that are triggered whenever a change to the user model contents occurs, they cause further updates to the model, one might view these as internal demons.

When a dialog episode triggers direct methods that change the user model, the changes trigger indirect methods which also change the user model. Indirect methods are implemented as after-methods on slots in the user model; to insure they get run whenever a slot change occurs.

Direct Methods The direct methods are an adaptation of related work on implicit user model acquisition in dialog advisory systems. The implicit acquisition rules developed in [Kass 88] are based on using natural language dialogs. Here dialog is used in a more general way. As previously described, it means any of the different interaction episodes that occur between a user and the system. Using this view it was possible to develop aset of direct implicit methods by modifying the implicature rules. These methods fall into four major categories of information available to the system through tracking dialogs with the user:

- techniques based on user decisions,
- methods triggered by information provided to the user,
- those triggered by optional actions on the part of the user, and
- ones activated when users access the hypertext information space.

Dialog episodes, between the user and LISP-CRITIC, terminate with a user decision (unless the session is aborted) to accept or reject the critic's advice. For either decision users requires the same type of knowledge (or level of understanding.) In both cases the system makes the same inference. A decision to accept a suggestion made by LISP-CRITIC causes the system to mark the rule behind the transformation as known to the user at level d2. A user's decision to reject a rule is handled similarly.

During dialog the system presents information to the user in the form of explanations, an event that triggers a direct inference that users know the explained entity. When the system explains a LISP concept, the level for that concept in the user model is marked as d2.

When users encounter explanations that are unsatisfactory they can access a hypertext information space as a fallback technique. Their selecting a mouse sensitive word is information that can be used to update the user model. The system can capture the selections and relate them to domain model entities where possible. When a match is found the system marks the domain entity at level d2 in the user model unless it is already at level d2, in which case its level is improved to d1. The assumption behind this method is an optimistic view that users actually read and understand information provided. When a user gets an explanation of the selected domain entity, there is an inference that they are familiar with that entity and their user model should now reflect that.

Direct methods change the user model based on explicit information observed in the dialog. These methods alone are inadequate for developing a useful sufficiently complete model in a reasonable amount of time. Additional methods that leverage this information, in the spirit of stereotypes, were needed. The structure of the domain model provides the basis for an additional class of methods that perform indirect implicit updating.

Indirect Methods Models of communications partners are based on more than the evidence provided directly during dialog. Computers, as Suchman pointed out [Suchman 87], do not have access to the rich set of information available to another human partner, so this research looked into ways to enrich the model by using available resources.The idea for indirect methods were developed when it was observed that the links in the domain model which capture prerequisite knowledge for the domain entities could provide a source for implicit acquisition. These prerequisite were established for use in explanation, but the idea of using them for implicit acquisition resulted from noting that they may tell us something about what the individual knows about the domain — in the spirit of the notion, used in the UMFE system, that user knowledge "propagates" through a set of concepts [Sleeman 84].

Linkages between entities represented in the structure of the domain model form the basis for the indirect update techniques in LISP-CRITIC. A change occurring in the representation for the user's domain knowledge can be used to trigger further changes to the user model based on the prerequisite knowledge for the entity just changed. The indirect methods infer how well those prerequisites domain entities are known and put that information into the appropriate slot in the user model. Indirect methods exist for each class of domain entity: LISP-CRITIC rules, LISP functions and LISP concepts.

There is a set of functions associated with each LISP-CRITIC rule, they are each linked to that rule via the *functions-in-rule* relationship in the domain model; they are the functions in either the left hand side or right hand side of the rule. Often, rules are also based upon certain LISP concepts in the domain model. When the level for a rule is changed in the user model, indirect methods modify what the user model has to say about how well the user knows these associated functions and concepts. If the rule has been set to level d2 our indirect methods infer that the functions in that rule are also known at level d2, and nothing is inferred about the concepts behind the rule. When the level of a rule is set to d1, both the functions in that rule and the concepts on which it depends are set to level d2 in the user model.

The prerequisite to understanding a LISP function are its *dependent-on* domain concepts. When the level at which a function is known is set to d2, its *dependent-on* concepts are also set to level d2. If the level of a function is set to d1, those concepts are also set to d1.

Concepts themselves are linked to one another via the *dependent-on* relationship. When the level of a concept is changed to d1 in a user model, its prerequisites in that user model are also set to d1. In the situation where the level of a concept is changed to d2 its *dependent-on* concepts are marked at d1 when they were previously marked d2 in the model. If those *dependent-on* concepts were not already in the model (conceptually marked d0) then they are added to it with a d2 marking.

The domain model implementation also contains *related* links that identify functions, concepts, or rules that are similar to another function, concept, or rule. These links could be used as the basis for a class of weak inference methods, such as predicting the ease with which a new entity could be introduced to the user.

The implemented indirect methods were designed conservatively; they are neither complete nor perfect. Their shortcomings, and indications about how to improve them came out during an evaluation; two results from this research are of general interest and utility:

- It is possible to define a class of implicit user model acquisition approaches that are indirect. These are leverage techniques that use information about users that is not directly observed, but is derived from other system knowledge (knowledge of a domain model, stereotypes, or etc) to indirectly enrich the models of those users.
- We can use the deep conceptual domain model that is needed for proper explanation as a source for a set of such indirect implicit user model acquisition methods.

The methodology followed provides an approach that can be used for developing user model acquisition techniques to serve other situations.

5.1.4 Access to the User Model

One objective for the user modelling component architecture was to insure that other system components can easily access the model. Another consideration was for the model to easily incorporate additional information. Significant theoretical issues or results were neither addressed or discovered in this aspect of the work, but played a role in selecting an object-oriented approach. Access methods support the current explanation component framework and providing information of value to other components.

The interface functions support the explanation strategies. The user model can be queried to determine which domain objects a user knows or does not know. Some example interface functions are ones that determine how well a user knows a domain entity (i.e., at what level), and whether a domain object was previously explained.

An attempt was made to conjecture the additional information a user model might be asked to provide, and include functions in the framework needed in those situations. Slots in the model record all rules-fired during previous dialog episodes and the number of times a user has invoked the critic. Other such information includes the user's goals and previous programming language experience. The goal can be acquired by explicit query of users during their initial session with LISP-CRITIC; currently it defaults to "simplifying" code to make the program easier for others to read and maintain. Previous programming experience in other languages can also be obtained with an initial information-gathering session or interactive questionnaire.

Some access capabilities are reserved for internal use of the user modelling component itself, the instance-slots, for example, can only be directly updated by the modelling component. The component receives information from other components about user actions or explanations, and determines how to use that information. It decides what additions or modifications to make to the user model and calls the appropriate internal methods. The user modelling component is notified when a session terminates normally (is not aborted) and a set of cleanup actions invoked. These functions save the user model's current contents in a file so that information is not lost when the user logs out or the system is rebooted, and can be used during subsequent use of LISP-CRITIC.

When LISP-CRITIC is called, the system determines whether the user has a model already loaded into the current environment; does not, but the system saved one during a previous session; or have not previously used LISP-CRITIC. In the first case, the system does nothing; in the second case, it loads the most recent version of the user model; and in the third case it initializes a model for this programmer.

5.2 LISP-CRITIC Explanation System

The explanation component shows conforms to the previously discussed framework for explanation giving. A focus in this implementation is to use information already available to the system, but to present that information so as to best support users' needs.

There are several sources of information that is already available to the system and which can satisfy some explanation requirements. This information is presented using techniques that were designed to provide users access to the information in four layers of increasing detail. These layers help to visualize how the system is designed and operates; they should not be confused with the conceptual levels shown in Figure 3-1.

1. A fundamental piece of information is the name of the rule that is the basis for a transformation. The rule name is an abstract reference to a chunk of domain knowledge, in the domain model that chunk is an instance of the class *lcr-rule*, and it may or may not have meaning to users. When it does have meaning, users may be satisfied just by knowing which rule fired and further explanations may not be required.
2. That second piece of information is the two versions of the code. The user can compare the system-transformed code with his own. The system displays the user's code together with the suggested changes. Sometimes this also triggers an understanding of the underlying concepts and rationale for the transformation.
3. The minimal-explanation layer is the point where empirical observations come into play. The user is provided with a text description of the system's advice based on the underlying concepts in the domain. The description is comprised of portions of hypertext associated with each domain concept and rule.
4. A hypertext-based information-space is also part of the underlying computational environment. LISP-CRITIC provides access to this information as a source of additional information for users who want to know more. Users navigate through the hypertext space after the system locates them within it in an appropriate context.

The explanations in Layers 3 and 4 need a user model to tailor their presentations to the individual user. The user modelling component can tell the explanation component which concepts users already understands so the system can avoid telling them what they already know. Layer 4 explanations back up the minimal-explanations with access to more detailed information.

The explanation approach is comprehensive and supports all four layers. An overview of the users' decision-making process, from the point the system makes a recommendation until they decide to accept or reject the suggested transformation is shown in a decision flow chart in Figure 5-2. The user, upon being informed that LISP-CRITIC recommends a particular change to his or her program, can get an explanation for that advice, or bypass the explanation completely, deciding right then to accept or reject the suggestion.

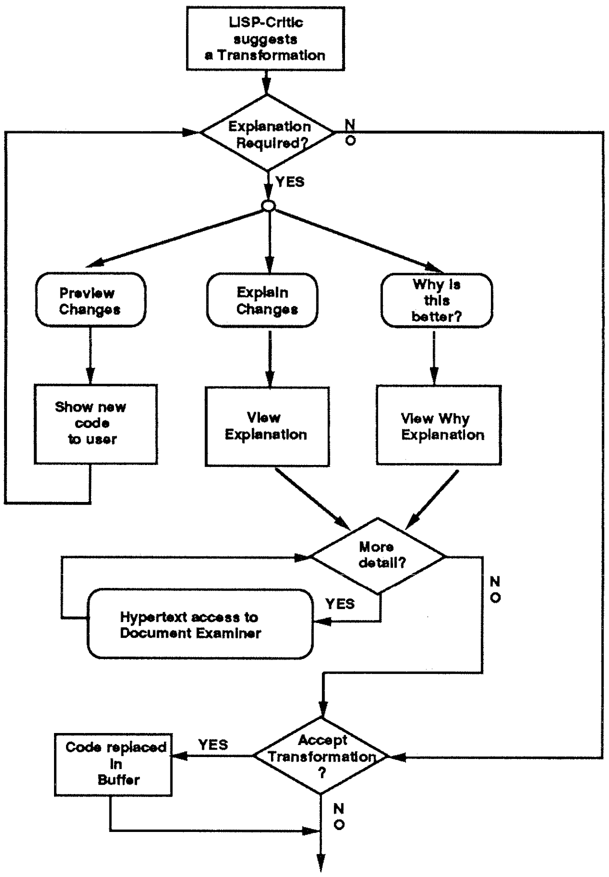
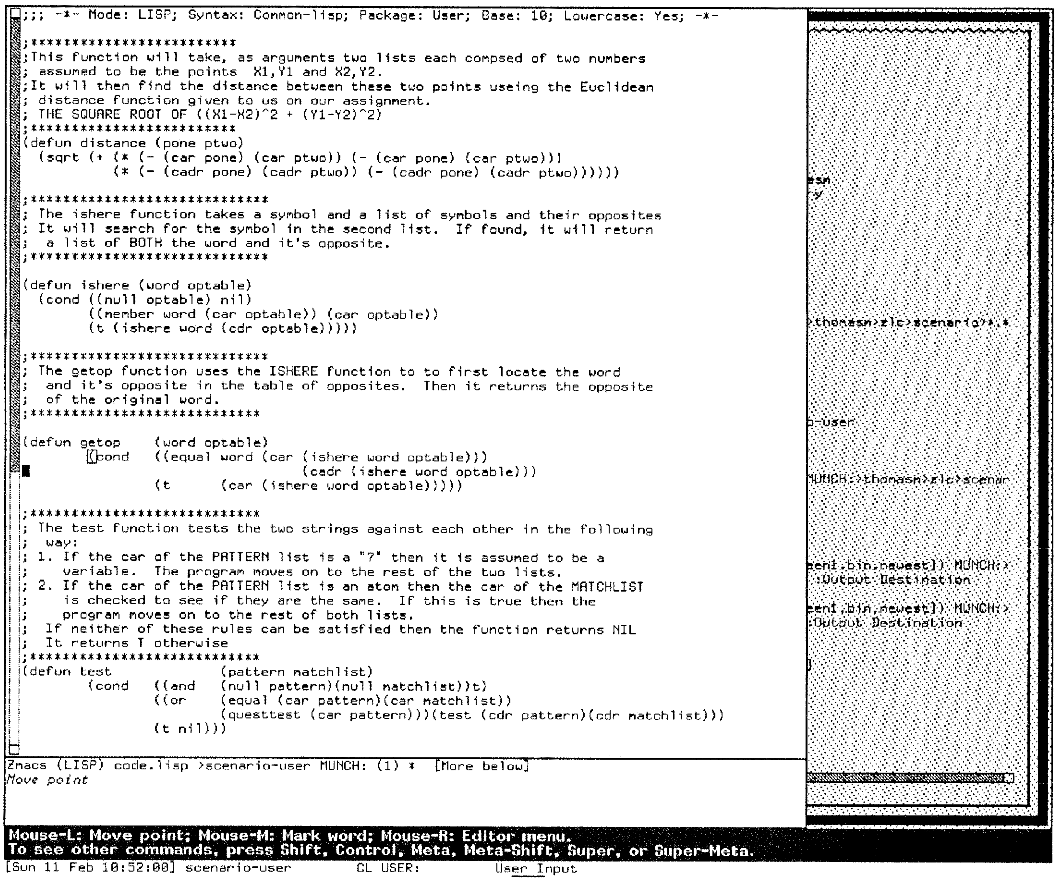


Figure 5-2: User Decision-Making Process in LISP-CRITIC

The explanation system does not attempt to present explanations as though they were generated by an intelligent agent, but rather uses combinations of straightforward, concise, prewritten sentences. What distinguishes this approach from most systems using canned-text is the role played by the user model in constructing an appropriate explanation. Each part of the explanation is chosen using the domain model, the user model, and the explanation strategies.

6. LISP-CRITIC Scenario

Here we describe a user interacting with LISP-CRITIC. The LISP code in this scenario was written by an undergraduate Computer Science student enrolled in an introductory artificial intelligence course[3] and comes from the corpus of programs used in the evaluation of the the user modelling component. It is the program developed for the student's first assignment in LISP. An initial user model (its partial contents can be seen in Figure 6-3) was provided to the system.



```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: User; Base: 10; Lowercase: Yes; -*-

;*************************
;This function will take, as arguments two lists each compsed of two numbers
; assumed to be the points  X1,Y1 and X2,Y2.
;It will then find the distance between these two points useing the Euclidean
; distance function given to us on our assignment.
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-Y2)^2)
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)) (- (car pone) (car ptwo)))
           (* (- (cadr pone) (cadr ptwo)) (- (cadr pone) (cadr ptwo))))))

;*****************************
; The ishere function takes a symbol and a list of symbols and their opposites
; It will search for the symbol in the second list.  If found, it will return
;  a list of BOTH the word and it's opposite.
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (car optable))
        (t (ishere word (cdr optable)))))

;*****************************
; The getop function uses the ISHERE function to to first locate the word
;  and it's opposite in the table of opposites.  Then it returns the opposite
;  of the original word.
;****************************

(defun getop    (word optable)
        (cond   ((equal word (car (ishere word optable)))
                                (cadr (ishere word optable)))
                (t      (car (ishere word optable)))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test             (pattern matchlist)
        (cond   ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))

Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *   [More below]
Move point

Mouse-L: Move point; Mouse-M: Mark word; Mouse-R: Editor menu.
To see other commands, press Shift, Control, Meta, Meta-Shift, Super, or Super-Meta.
[Sun 11 Feb 10:52:00] scenario-user        CL USER:        User Input
```

User editing LISP code in the ZMACS buffer.

Figure 6-1: Scenario-User's LISP Program

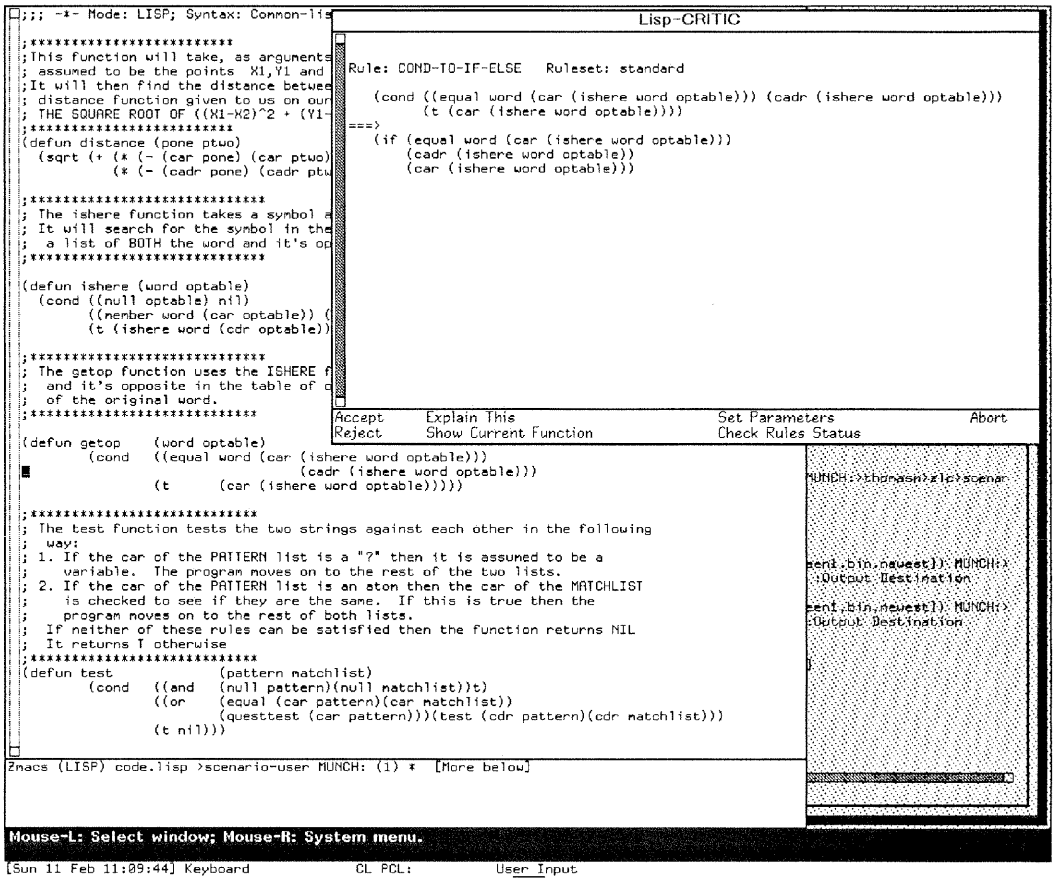
LISP-CRITIC was not used previously by this student programmer, therefore the startup user model is based on responses to a data collection questionnaire completed by him, augmented with information about concepts explained in class. The theoretical investigations of user modelling in this research concentrated on methods for enhancing an existing model using the context of the user-system dialog, assuming the existence of some sort of initial or start-up model of each user. The rationale for this assumption is the existence of several available techniques for providing the initial model (interactive questioning of users, stereotyping, classification categories, etc) that could be adapted for use in LISP-CRITIC. It was felt that rather than attempting to implement the entire range of methods that first build a model "from

[3]This student, whose identity is not revealed, happens to be male; therefore, in this discussion *he* will be referred to using male pronouns.

scratch" and then improve it over time, the work should concentrate on the more difficult and less well understood problem of how to enhance that model over time (dynamically).

6.1 Initial Dialog Episode

The initial screen image of the user working on his code in ZMACS is shown in Figure 6-1. He wrote a program using the ZMACS editor on a Symbolics LISP Machine. From the editor he invokes LISP-CRITIC using a HYPER-S key combination. LISP-CRITIC examines a single function definition (*defun*) at a time. That function definition is identified by the system as the one within which the user has currently positioned his cursor. For first scenario episode it is the *defun* for *getop*. The figure shows the entire buffer to emphasize that LISP-CRITIC recognizes the user's context (from the cursor), just as a knowledgeable human assistant might; the programmer does not have to scroll the window to a particular configuration or mark a section of the program to identify it to the critic.



```
;;; -*- Mode: LISP; Syntax: Common-lis
;*************************
;This function will take, as arguments
; assumed to be the points  X1,Y1 and
;It will then find the distance betwee
; distance function given to us on our
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)
           (* (- (cadr pone) (cadr ptw

;*****************************
; The ishere function takes a symbol a
; It will search for the symbol in the
;  a list of BOTH the word and it's op
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (
        (t (ishere word (cdr optable))

;*****************************
; The getop function uses the ISHERE f
;  and it's opposite in the table of o
;  of the original word.
;****************************

(defun getop    (word optable)
        (cond   ((equal word (car (ishere word optable)))
                                (cadr (ishere word optable)))
                (t      (car (ishere word optable)))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test             (pattern matchlist)
        (cond   ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))
```

```
Rule: COND-TO-IF-ELSE   Ruleset: standard

   (cond ((equal word (car (ishere word optable))) (cadr (ishere word optable)))
         (t (car (ishere word optable))))
===>
   (if (equal word (car (ishere word optable)))
       (cadr (ishere word optable))
       (car (ishere word optable)))
```

```
Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More below]
```

```
MUNCH:>thomasn>zlc>scenar
ent.bin.newest]) MUNCH:>
 :Output Destination
ent.bin.newest]) MUNCH:>
:Output Destination
```

```
[Sun 11 Feb 11:09:44] Keyboard          CL PCL:          User Input
```

LISP-CRITIC is accessed and makes a suggestion on how to improve the user's code.

Figure 6-2: Scenario-User Invokes LISP-CRITIC on Function *getop*

LISP-CRITIC examines the user's code for ways to simplify it. In this case it finds that a *cond* special form could be replaced by an *if* special form and makes that suggestion, as shown in Figure 6-2. The user has the choices in the menu bar at the bottom of the LISP-CRITIC window, of interest here are the options to *accept*, or *reject* LISP-CRITIC's suggestion, or to ask the system to *explain this*. The user does not understand the suggestion, so he selects the *explain this* menu option. The system calls the explanation component which obtains from the domain model the concepts required to understand this rule. Then the explanation component calls the user modelling component to determine which aspects of that knowledge the user lacks.

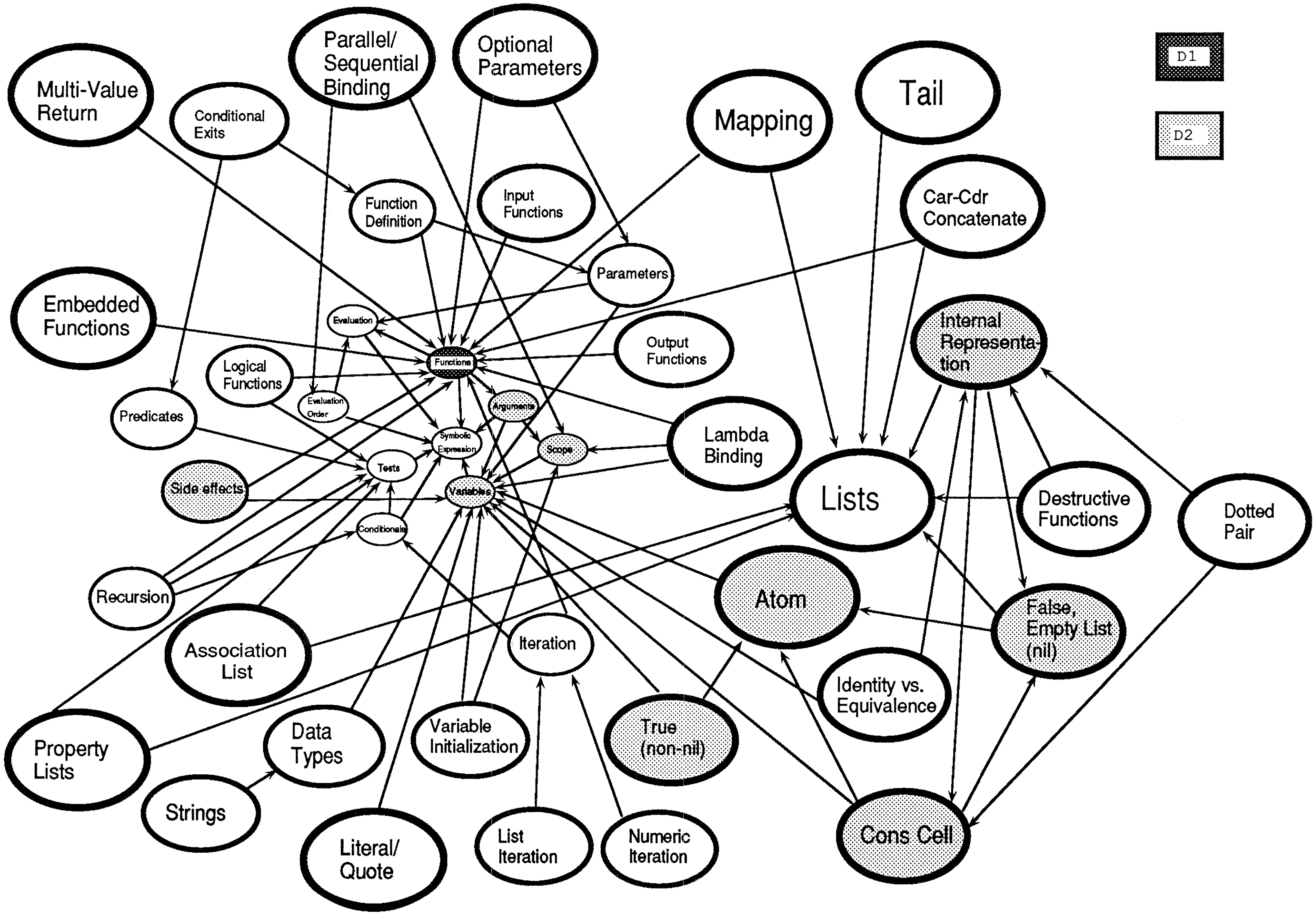
The user model is conceptually a coloring of the domain model graph, that the graph has concept, function, and LISP-CRITIC rule layers. Determining what to explain to a user involves extracting information from the user model, the appropriate concepts required to understand a transformation. Figure 6-4 shows the initial coloring of the concept layer for the scenario-user. Concepts known to the user model are shaded appropriately, depending on whether they are in d1 or d2. Unshaded concepts are at level d0. An equivalent textual representation of the domain knowledge slots for the user model is displayed in Figure 6-3.

```
Summary data for user model for SCENARIO-USER

Following concepts in D1

FUNCTIONS

Following concepts in D2
INTERNAL-REPRESENTATION
SIDE-EFFECTS
CONS-CELL
VARIABLES
SCOPE
LISP-ATOM
ARGUMENTS
FALSE/EMPTY-LIST/NIL
TRUE/NON-NIL

Following functions in D1

Following functions in D2

Following lcr-rules in D1

Following lcr-rules in D2

Rules-fired by name and number of firings
NIL
```

This is the state of the user model at the beginning of the scenario. The concepts came from user's self-ratings on the initial questionnaire.

Figure 6-3: Initial User Model

The domain model begins with the *cond-to-if-else* rule and, using the links between domain model entities, accumulates a *concept set* consisting of all prerequisites to understanding the rule. The user modelling component filters the concept set so the explanation component can focus on explaining only those concepts which users do not know. When LISP-CRITIC is invoked, the user's code is examined for ways to simplify it. In the first scenario, the system recommended that *cond* could be replaced by the *if* special form, Figure 6-2. Before the user decided whether to accept or reject LISP-CRITIC's suggestion, he asked for an explanation. The system then determined what was required in order to understand this rule, and the aspects of that knowledge that the user lacks.

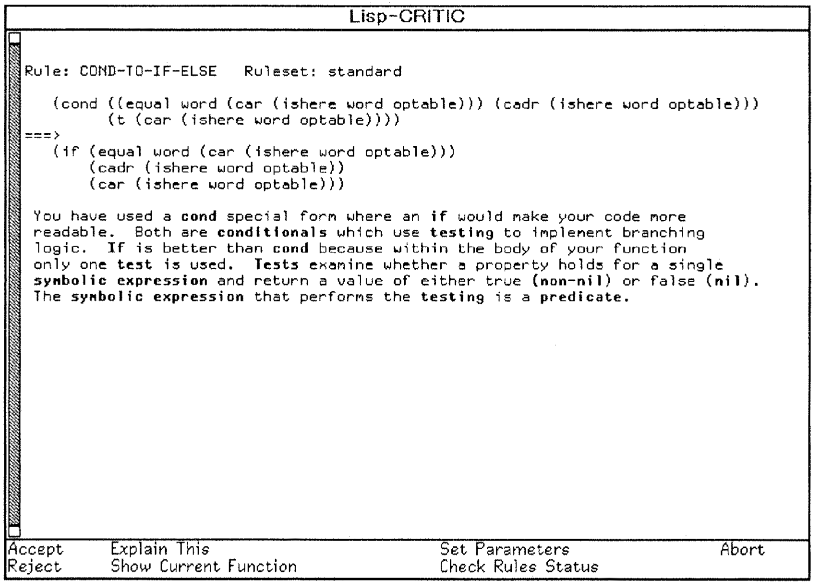


This graph is colored to represent a the concept portion of the initial knowledge state of the user in the scenario. This information is also shown textually in Figure 6-3.

Figure 6-4: Coloring of Conceptual Graph for Initial User Model

The explanation component interrogated the domain model and was provided a list of concepts underlying the *cond-to-if-else* rule. By traversing the domain model beginning at the node representing the *cond-to-if* rule, and using the *dependent-on* links between domain model objects the system accumulates the *concept set*. For the *cond-to-if-else* rule in the first episode of the scenario, (FIGURES 6-2 THROUGH 6-6) traversal of the domain generated a *concept set* of 13 items: *lists, symbolic-expression, evaluation, tests, variables, conditionals, scope, predicates, lisp-atom, arguments, false/empty-list/nil, true/non-nil, and functions.*

That set was personalized for the user in the scenario, to determine the subset of concepts to actually be explained. There are three levels d1, d2, and d3 at which a user can understand a given concept. The current implementation captures users' knowledge in terms of concepts that are well known to the user (d1), known to the user but not well (d2), and unknown (d0). For the concept set, the user model indicated (by their absence from the concepts-known slot in the user model) that the user has no knowledge (level d0) of six of them: *predicates, conditionals, tests, evaluation, symbolic-expression, and lists.* It indicated, based on their markings in the concepts-known slot, some knowledge (d2) of 6 others: *true/non-nil, false/empty-list/nil, arguments, lisp-atom, scope and variable*; and good knowledge (level d1) about just one, *functions.* That information was provided to the explanation component in three sublists, one each for d0, d2, and d1.



```
Lisp-CRITIC

Rule: COND-TO-IF-ELSE   Ruleset: standard

   (cond ((equal word (car (ishere word optable))) (cadr (ishere word optable)))
         (t (car (ishere word optable))))
===>
   (if (equal word (car (ishere word optable)))
       (cadr (ishere word optable))
       (car (ishere word optable)))

 You have used a cond special form where an if would make your code more
 readable.  Both are conditionals which use testing to implement branching
 logic.  If is better than cond because within the body of your function
 only one test is used.  Tests examine whether a property holds for a single
 symbolic expression and return a value of either true (non-nil) or false (nil).
 The symbolic expression that performs the testing is a predicate.
```

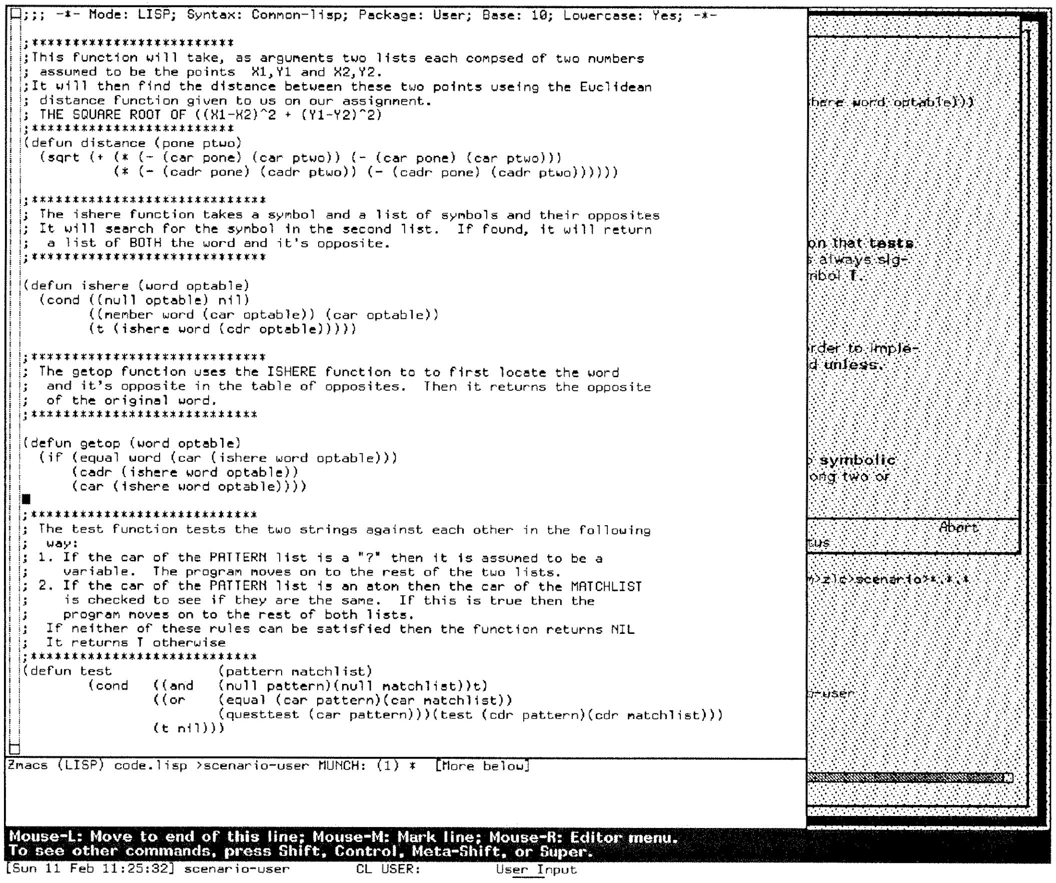
LISP-CRITIC explains a suggestion based on the *cond-to-if* rule to include those prerequisite concepts that the user does not know.

Figure 6-5: Explanation For *cond-to-if-else* Rule

Furthermore, within each sublist, concepts were ordered according to an implicit hierarchy captured in the *dependent-on links* in the domain model. The explanation component can ultimately use this information to reason about how to generate an explanation for the user, but the current implementation, using a simplified strategy for testing purposes only, selects the first three concepts in that filtered list *predicates, conditionals, and tests.* Mock explanations using these three concepts as a basis are displayed in Figure 6-5. Ideally, the user finds the explanation adequate; but other concepts fundamental to understanding, or related to, these concepts are shown as mouse-sensitive objects displayed in bold.

Selecting any of them will display either an explanation associated with that object from the domain model, or a description from the Symbolics Document Examiner.

The user accepts this suggestion, and LISP-CRITIC automatically rewrites the modified portion of the user's code in the editing buffer. In Figure 6-6 the body of function *getop* has been changed to reflect the *cond-to-if-else* transformation. In the function definition (*getop*), LISP-CRITIC found only one transformation to suggest, therefore at this point the programmer is returned to his code editing buffer and LISP-CRITIC's window becomes inactive; it moves into the background, out of the programmer's view.



```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: User; Base: 10; Lowercase: Yes; -*-

;*************************
;This function will take, as arguments two lists each compsed of two numbers
; assumed to be the points  X1,Y1 and X2,Y2.
;It will then find the distance between these two points useing the Euclidean
; distance function given to us on our assignment.
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-Y2)^2)
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)) (- (car pone) (car ptwo)))
           (* (- (cadr pone) (cadr ptwo)) (- (cadr pone) (cadr ptwo))))))

;*****************************
; The ishere function takes a symbol and a list of symbols and their opposites
; It will search for the symbol in the second list.  If found, it will return
;  a list of BOTH the word and it's opposite.
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (car optable))
        (t (ishere word (cdr optable)))))

;*****************************
; The getop function uses the ISHERE function to to first locate the word
;  and it's opposite in the table of opposites.  Then it returns the opposite
;  of the original word.
;****************************

(defun getop (word optable)
  (if (equal word (car (ishere word optable)))
      (cadr (ishere word optable))
      (car (ishere word optable))))
■
;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test             (pattern matchlist)
        (cond   ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))

Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More below]

Mouse-L: Move to end of this line; Mouse-M: Mark line; Mouse-R: Editor menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Sun 11 Feb 11:25:32] scenario-user       CL USER:        User Input
```

After the user accepts LISP-CRITIC's suggestions, the system modifies the code in his buffer.

Figure 6-6: Modified ZMACS Buffer

The user's actions throughout this episode trigger changes in his user model. During the dialog episode, the user was satisfied with the explanation and *accepted* this suggestion. The content of this dialog episode was used to update the user model. The cues from this episode which are important to the user modelling component are: the receipt of explanations about certain concepts (e.g. *conditionals, predicates* and *tests*) and the user deciding to accept the *cond-to-if* transformation. Cues triggered direct inferences that changed the user model and these changes in turn triggered indirect inferences that will be explained in the next section. A portion of the updated model is shown textually in Figure 6-7, and its associated graph coloring for the concepts layer in Figure 6-8. The system's design incorporates techniques that recognize that a model has been constructed for this user and makes use of that version of in subsequent dialogs between LISP-CRITIC and this user.

```
                    Summary data for user model for SCENARIO-USER

Following concepts in D1                    Following functions in D1
FUNCTIONS
                                            Following functions in D2
Following concepts in D2                    IF
SYMBOLIC-EXPRESSION                         COND
LISTS
EVALUATION                                  Following lcr-rules in D1
TESTS
CONDITIONALS                                Following lcr-rules in D2
PREDICATES                                  USER::COND-TO-IF-ELSE
INTERNAL-REPRESENTATION
SIDE-EFFECTS                                Rules-fired by name and number of firings
CONS-CELL
VARIABLES                                   USER::COND-TO-IF-ELSE
SCOPE                                         TIMES-FIRED    1
LISP-ATOM                                     TIMES-ACCEPTED 1
ARGUMENTS                                     TIMES-REJECTED 0
FALSE/EMPTY-LIST/NIL
TRUE/NON-NIL
```

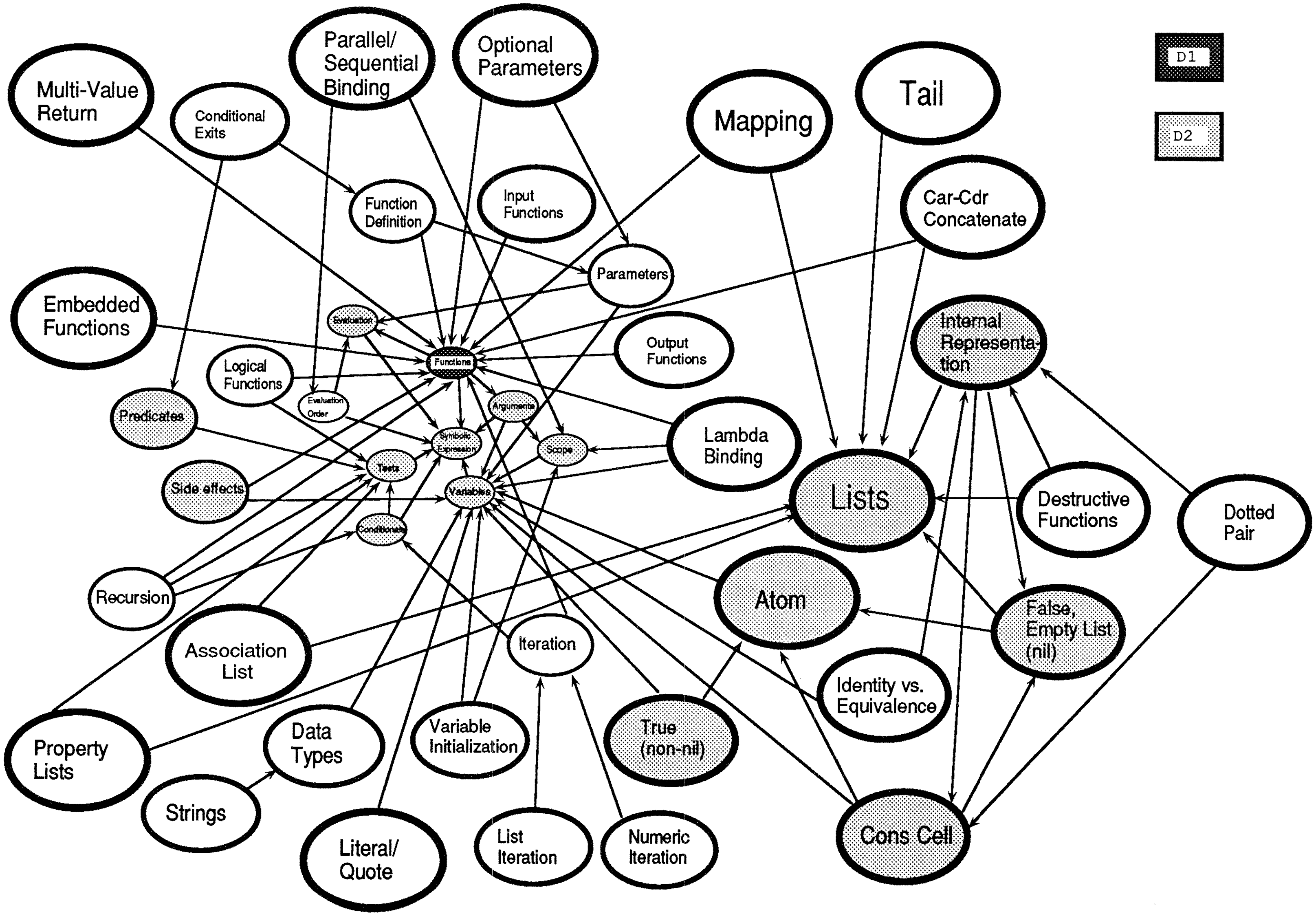
The contents user of the model after the first dialog episodes. Changes to the content, when compared to Figure 6-3, are a result of user actions during the episode triggering inference methods that update the model.

Figure 6-7: User Model Contents after First Dialog Episode

6.2 Second Dialog Episode

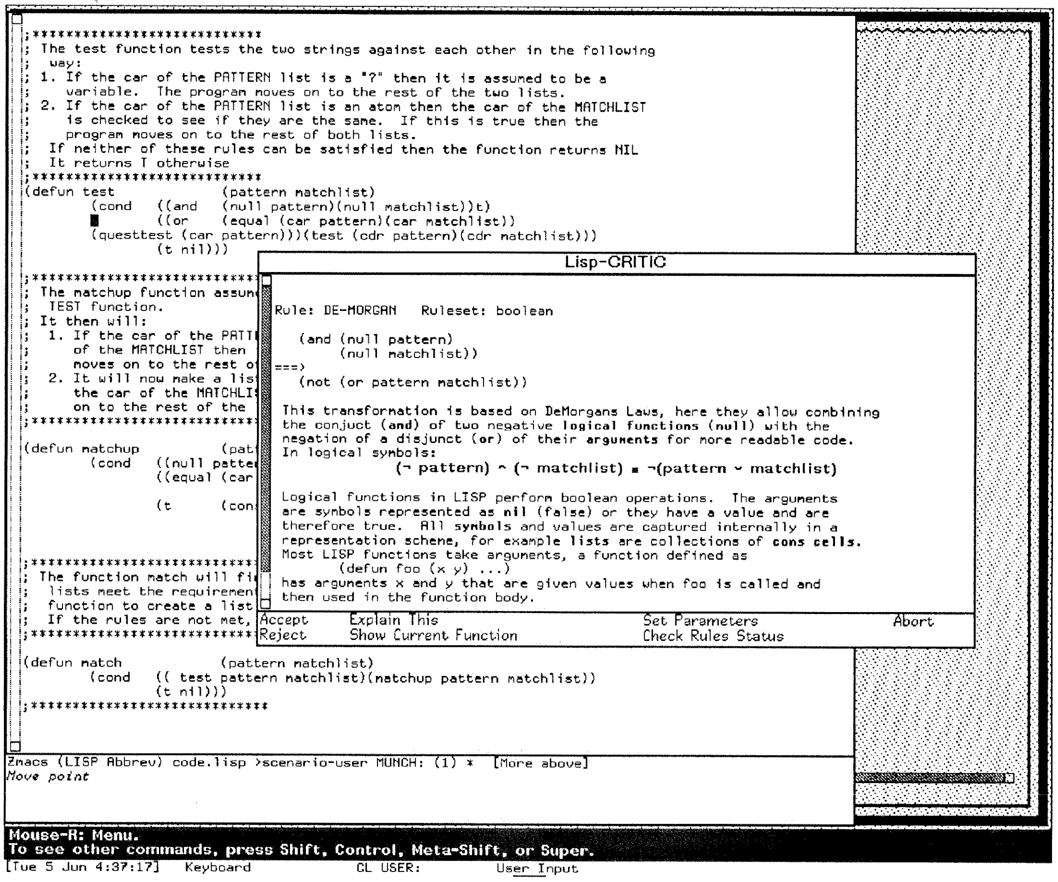
The second scenario episode could occur immediately, or at a later time; the user model is saved between sessions and reused when the user subsequently accesses LISP-CRITIC. Our user now requests LISP-CRITIC to look over the code written for function *test* which causes a recommendation based on a rule *de-morgan* (the rule applies DeMorgan's law from logic to combine booleans) and again he requests an explanation. The explanation component once again consults the domain and user models to determine what needs to be explained to this particular user. Those top three concepts selected by the simplified explanation strategy are *logical functions*, *internal representation*, and *arguments*; these are integrated into the complete text of the mock explanation shown in Figure 6-9.

When the user accepted the suggestion, his user model was again updated. The system suggested a second transformation for this function, one based on the *cond-erase-pred.t* rule; it was explained in Figure 6-12 based once again on information from the user model about how well the user knows the underlying concepts for this rule. It happens in this case that the concepts incorporated into the explanation belong in the user's d2 level because none of the *concept set* underlying that rule were unknown (at level d0). A third suggestion, based on the *cond-erase-t.nil* rule triggered and the user accepted it, but without an explanation because the prerequisite concepts are similar to those already explained for the previous rule. The user model was dynamically updated throughout the dialog. Each time information is shown or a decision made, inference methods triggers. The domain knowledge portion of the user model at the conclusion of the scenario is shown in Figure 6-10; its associated graph coloring for the concept layer in Figure 6-11.



This graph is colored to represent the concept knowledge of the user after the first dialog episode. The content information is also shown textually in Figure 6-7.

Figure 6-8: Coloring of Conceptual Graph for User Model after First Dialog Episode



```
;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test               (pattern matchlist)
       (cond   ((and   (null pattern)(null matchlist))t)
       █       ((or    (equal (car pattern)(car matchlist))
       (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
               (t nil)))

;****************************
; The matchup function assum
;  TEST function.
; It then will:
;  1. If the car of the PATT
;     of the MATCHLIST then
;     moves on to the rest o
;  2. It will now make a lis
;     the car of the MATCHLI
;     on to the rest of the
;****************************

(defun matchup            (pat
       (cond   ((null patte
               ((equal (car

               (t      (con



;****************************
; The function match will fi
;  lists meet the requiremen
;  function to create a list
;  If the rules are not met,
;****************************

(defun match              (pattern matchlist)
       (cond   (( test pattern matchlist)(matchup pattern matchlist))
               (t nil)))
;****************************
```

```
Lisp-CRITIC

Rule: DE-MORGAN   Ruleset: boolean

   (and (null pattern)
        (null matchlist))
===>
   (not (or pattern matchlist))

 This transformation is based on DeMorgans Laws, here they allow combining
 the conjuct (and) of two negative logical functions (null) with the
 negation of a disjunct (or) of their arguments for more readable code.
 In logical symbols:
               (¬ pattern) ∧ (¬ matchlist) ≡ ¬(pattern ∨ matchlist)

 Logical functions in LISP perform boolean operations.  The arguments
 are symbols represented as nil (false) or they have a value and are
 therefore true.  All symbols and values are captured internally in a
 representation scheme, for example lists are collections of cons cells.
 Most LISP functions take arguments, a function defined as
          (defun foo (x y) ...)
 has arguments x and y that are given values when foo is called and
 then used in the function body.
Accept    Explain This                     Set Parameters            Abort
Reject    Show Current Function            Check Rules Status
```

```
Zmacs (LISP Abbrev) code.lisp >scenario-user MUNCH: (1) *  [More above]
Move point

Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 5 Jun 4:37:17]   Keyboard          CL USER:         User Input
```

LISP-CRITIC is asked to examine another part of the user's program, a suggested transformation and its explanation is shown.

Figure 6-9: Explanation For *de-morgan* Rule

Our scenario user also accepts this suggestion. A third rule, *cond-erase-t.nil*, triggers; it is very similar to the previous rule therefore, the user accepts LISP-CRITIC's recommendation without explanation. He is able to generate his own explanation because he knows a similar rule that was previously encountered, and is familiar with the underlying concepts. Throughout the dialog, the user model is updated each time the user receives an explanation or makes a decision.

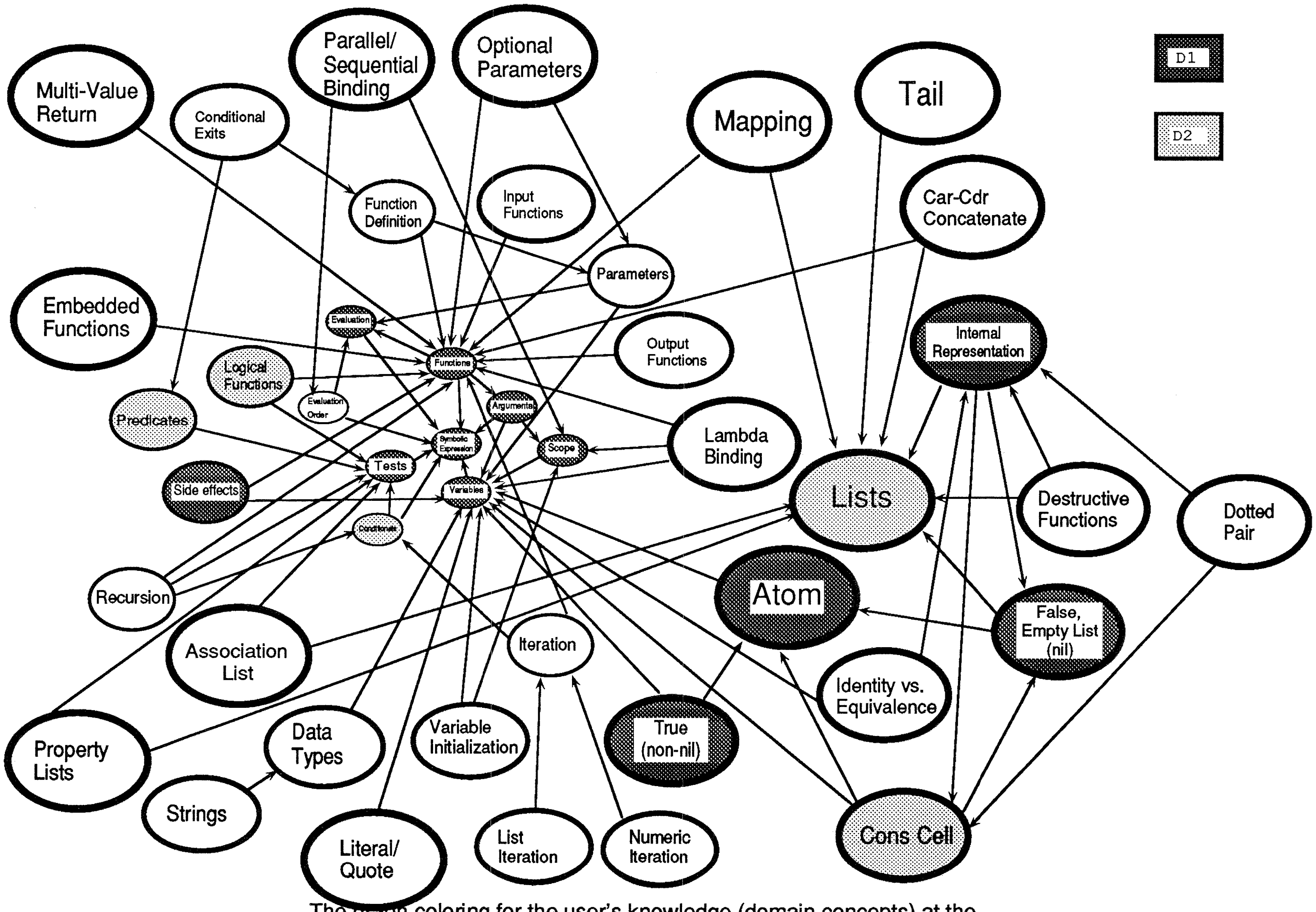
The development of the explanation component has not progressed past the conceptual and methodological stages. The explanations shown in this scenario are only intended to point out the relationship between the work on the domain and user modelling and the requirements for explanation-giving. As presented, the explanations do not constitute a finished product and should be viewed primarily as a means for understanding how the system makes choices of what to present during its dialogs with the user. Additional work to implement the presentation strategies for explanation-giving is required. This is not a trivial problem; it is one of the three major issues in explanation identified in [Chandrasekaran, Tanner, Josephson 89].

```
            Summary data for user model for SCENARIO-USER

Following concepts in D1           Following lcr-rules in D1
SYMBOLIC-EXPRESSION
EVALUATION                         Following lcr-rules in D2
TESTS                              USER::COND-ERASE-T.NIL
INTERNAL-REPRESENTATION            USER::COND-ERASE-PRED.T
SIDE-EFFECTS                       USER::DE-MORGAN
VARIABLES                          USER::COND-TO-IF-ELSE
SCOPE
LISP-ATOM                          Rules-fired by name and number of firings
ARGUMENTS
FALSE/EMPTY-LIST/NIL               USER::COND-ERASE-T.NIL
TRUE/NON-NIL                         TIMES-FIRED    1
FUNCTIONS                            TIMES-ACCEPTED 1
                                     TIMES-REJECTED 0
Following concepts in D2
LOGICAL-FUNCTIONS                  USER::COND-ERASE-PRED.T
LISTS                                TIMES-FIRED    1
CONDITIONALS                         TIMES-ACCEPTED 1
PREDICATES                           TIMES-REJECTED 0
CONS-CELL
                                   USER::DE-MORGAN
Following functions in D1            TIMES-FIRED    1
                                     TIMES-ACCEPTED 1
Following functions in D2            TIMES-REJECTED 0
NULL
NOT                                USER::COND-TO-IF-ELSE
OR                                   TIMES-FIRED    1
AND                                  TIMES-ACCEPTED 1
IF                                   TIMES-REJECTED 0
COND
```

The state of the user model after the second (final) dialog episode.

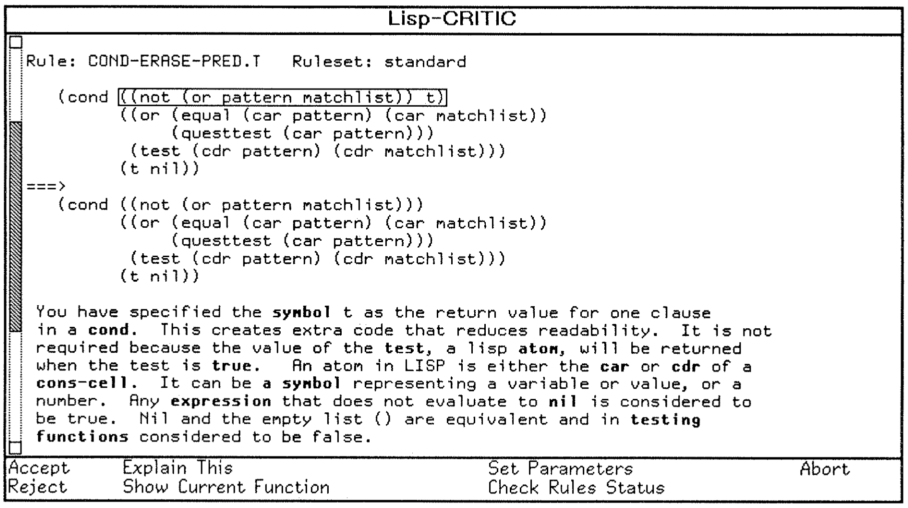
Figure 6-10: User Model Contents after Second Dialog Episode

To show how the direct and indirect acquisition techniques work together, let us review a portion of the scenario. The concept *tests* is marked at the d2 level in the user model shown in Figure 6-8. This resulted from a direct inference based on a method that claims when a domain entity is explained to a user, that user is now aware of its existence and has a fundamental understanding of it — the user knows of the concept but is not proficient in applying it in every circumstance. The domain model also tells us that, for the concept *tests*, a prerequisite (according to the *dependent-on* links in the domain model) is the concept *symbolic-expression*. Therefore, an indirect inference places *symbolic-expression* at level d2 for this user. Similar direct and indirect user model acquisition methods fire for the *cond-to-if-else* rule, its underlying functions, and its *dependent-on* concepts. More domain entities in the user model in Figures 6-7 and 6-8 get marked at the d2 or d1 level as the result of indirect implicit methods than as a result of direct methods.



The graph coloring for the user's knowledge (domain concepts) at the end of the scenario. The textual representation is shown in Figure 6-10.

Figure 6-11: Coloring of Conceptual Graph for User Model after Second Dialog Episode



```
Rule: COND-ERASE-PRED.T   Ruleset: standard

  (cond ((not (or pattern matchlist)) t)
        ((or (equal (car pattern) (car matchlist))
             (questtest (car pattern)))
         (test (cdr pattern) (cdr matchlist)))
        (t nil))
===>
  (cond ((not (or pattern matchlist)))
        ((or (equal (car pattern) (car matchlist))
             (questtest (car pattern)))
         (test (cdr pattern) (cdr matchlist)))
        (t nil))

 You have specified the symbol t as the return value for one clause
 in a cond.  This creates extra code that reduces readability.  It is not
 required because the value of the test, a lisp atom, will be returned
 when the test is true.   An atom in LISP is either the car or cdr of a
 cons-cell.  It can be a symbol representing a variable or value, or a
 number.  Any expression that does not evaluate to nil is considered to
 be true.  Nil and the empty list () are equivalent and in testing
 functions considered to be false.
```

LISP-CRITIC recommends a second transformation in the *defun* for test, which the user asks to be explained.

Figure 6-12: Explanation For *cond-erase-pred.t* Rule

7. Summing It Up and Conclusions

The general scheme of this research was to study user modelling research in other areas; to develop an understanding of what is required for a user model to support cooperative problem solving; and, from those analyses, to develop an approach for supporting a computer-based critic. A user model that meets those requirements was implemented for LISP-CRITIC, and subjected to an evaluation. The results of the system development work and analysis suggested ideas about the generalizability of the methodology, and indicated possible extensions to the approach as well as directions for future research.

7.1 Summary

This research was accomplished in a context of developing a paradigm for cooperative problem solving, and in the context of knowledge-based systems that support user learning. In principle all cooperative systems should also support learning. Users need access to system-provided explanations in order for that learning to take place. Furthermore, those explanations should be tailored to their individual expertise in the application domain. This need for individualized explanations motivates a requirement for idiosyncratic user models. These characteristics of knowledge-based computer systems, that they support collaborative human-computer effort, and also, that they provide learning opportunities, determine the general requirements for the user modelling approach.

Critiquing is one way to use computer knowledge bases to aid users in their work and at the same time support their learning needs. Critiquing systems, also called critics, are an alternative to traditional experts systems. The generality of the approach is demonstrated by a survey of the literature which shows that critiquing has been successfully used in diverse application domains. In order to enhance current critiquing approaches so that these systems move from simple "suggestors" of how to improve a user's work, to ones which can interact with them in a collaborative style, will require models of users. These models will help systems adapt explanations of their domain knowledge to individual users. Critics are not the only approach to building better knowledge-based systems, but a growing number of such systems will contain a critiquing component. Some of them need detailed understanding of users' problems, tasks and goals; but more commonly they will have limited yet helpful capabilities, one of which is to model the knowledge of individual users.

As a result of studying related research in user modelling. The approach used includes an architecture for a user modelling component comprised of a representation scheme for the models, acquisition techniques, and methods for accessing the models. The methodology followed was to identify the requirements for a user modelling component to support explanations based on a theoretical model of users' expertise, a conceptual model of the domain, the need to acquire the model using implicit methods, all the while keeping in mind a goal of generality. The resulting conceptual architecture was instantiated in a specific system.

LISP-CRITIC provides a suitable context for investigating user modelling because integrating a user modelling component to support explanation giving was a natural extension of previous work. A domain model was required to support both explanation-giving and user modelling. It links the system's operational knowledge (LISP-CRITIC rules) to the domain knowledge necessary for explanation-giving and representing users' expertise. An analysis of LISP determined what to represent in the domain model, and the appropriate techniques for achieving that representation. The implemented domain model captures knowledge of LISP in a conceptual structure. From our analysis of the domain it was determined that the fundamental domain entities are LISP concepts, LISP functions and LISP-CRITIC rules, they are all interconnected via semantic relationships. The domain model was implemented using the Common LISP Objects System (CLOS).

Cooperative knowledge-based systems take advantage of the different strengths of users and computer systems. Computers are potential sources of expert domain knowledge and can be used to make suggestions; their role must also include the ability to explain those suggestions. Explanation systems often fail because they are based on implicit assumptions that explaining is a one-shot affair, and that artificially intelligent systems will be able to retrieve or produce complete and individualized text. Another approach is to take advantage of information and present computer technology. The explanation approach focuses on determining which concepts to explain to a user rather than on choosing a prestored explanation. Executing that process requires a domain model that can provide the set of concepts needed in a given explanation situation, and a user model that can help tailor the explanation to a given individual.

The approach provides four layers of explanation that can be accommodated in LISP-CRITIC. The first two layers are not explanations in the strictest sense but rather techniques for presenting the critic's advice that facilitate user understanding; they are detailed descriptions of what the system suggests. Rhetoric principles and discourse comprehension research provide foundations for a minimal approach that make-up the 3rd layer. Such minimal explanations are guided by a domain and user model that provide, to the system, information about what needs to be explained in order for a user to understand a particular domain entity. The highest layer is a rich hypertext information space that provides a fallback capability for situations in which users need more details. In that hypertext space, users can investigate LISP functions or examine concepts that they still do not understand.

The user modelling component developed for LISP-CRITIC represents what the system knows about each user in an object oriented structure, acquires those user models, provides access to them and retains them for future use. The user model is also implemented in CLOS. The design objectives were based on what is required to support explanation-giving; these objectives guided specification of the architecture for the user modelling component.

Representation of the model is an enhancement of overlay modelling techniques. The approach captures the domain entities a user knows, a subset of those in the domain model and also marks them in accordance with how well they are known. Conceptually, there is a coloring of the domain model graph unique to each individual.

Other system components access the individual models via a set of generic interface functions. In this research, access for the explanation component has been emphasized; but we have attempted to make the methods general so that current system components, such as the critiquing engine, or even new components that are added, like a tutor, can make use them.

The acquisition subcomponent contains direct methods that make use of episodes in the user-computer dialog. These are "evidence-based" methods based on the observation that the interaction context contains useful information for inferring the knowledge state of a user. The subcomponent also contains indirect methods that are triggered by changes to individual user models. One outcome of the LISP-CRITIC system development and implementation work is a framework of user model acquisition techniques.

The development of the implicit methods is a significant contribution. An evaluation of their effectiveness and possible modifications was undertaken. The acquisition methods were evaluated in two ways. Programs written by students learning LISP were processed by LISP-CRITIC; and the individual user models for each programmer compared with one another, attending particularly to the changes that took place over time. The user models developed for each student were correlated with questionnaires assessing the students' expertise according to the topology of the domain model; the questionnaires were completed before each programming assignment. The evaluation showed that the user models conform to expectations about how user knowledge might change under the conditions these programs were produced. The model's contents were modified by the acquisition methods in a manner similar to what was expected: the models became more detailed as the system was exposed to more of the users' work; and they captured new concepts as students learned them during the course of completing three programming assignments. The evaluation pointed out opportunities for improving the acquisition methods. It also resulted in the observation that using a startup or initial model would probably improve the models' fidelity.

To achieve a completely operational model will require significant additional development and extensive testing. One limitation is that any user model is at best an "approximation" of a user's knowledge state, therefore, it will be difficult to determine when an acquisition methodology is as complete as possible. An outcome of this work is an awareness that a comprehensive user modelling system will be extremely complex; the problem will not yield to singular solutions or simple methods alone. Research efforts to date have tackled only a parts of the problem, usually in isolation. A complete implementation will have to integrate multiple techniques (e.g., stereotyping, explicit questioning and implicit acquisition methods) with detailed domain and perhaps task models. The work undertaken here focused on developing a domain model and implicit acquisition techniques.

The acquisition framework is a major contribution; it provides a pretheoretic scheme which can be used for developing user modelling components for human-computer interaction systems, and can serve as a guide for future research. The methodology followed here can be used for developing user models for applications other than critiquing; and to extend those critics already developed in other research. In this research the approach used was specifically developed for critiquing, but it provides a starting point for individualizing a more general class of cooperative problem solving systems. There are possibilities to share both the model in its current form with other interaction approaches, like advising or tutoring, and to use the methodology as a guide for developing new models to serve a range of applications, critiquing computer programs being just one them.

7.2 Conclusions

This project contributes to research in user modelling, explanation-giving, and cooperative knowledge-based systems. The use of a common deep conceptual domain representation for both explanation generation and user modelling is unique. Using the inherent structure of that deep domain model to perform implicit acquisition is a technique that enhances a system's ability to build up more complete idiosyncratic models of users. The explanation process begins with a single piece of procedural system knowledge, e.g., a rule that a user wants described. It serves as a starting point to extract the appropriate domain concepts; these are filtered through the user model and some of them are eventually explained to the user. This approach could potentially be used in a large class of human-computer interaction systems but depends upon domain and user models to inform the process. This work is an implementation of a model of users' domain expertise in a critiquing system. The possibilities and limitations uncovered here can aid developers of other computer-based critics. The framework for user modelling acquisition methods proved useful in developing a specific user modelling component, and it can be used to guide design analysis and architectural specification for others.

It would be premature to claim that a general theory of user modelling is forthcoming, but this effort provides a better understanding about some significant aspects of such a theory. Specifically, we now know more about what is required of a user model that supports explanation-giving; the sort of techniques an interactive system can use to implicitly acquire such a model; and how a concept-based domain model can serve as a basis for user model representation, and at the same time support user model acquisition. These ideas expose a new range of issues and directions for research into user modelling that may eventually provide general methods able to accommodate a broad class of human-computer interaction systems.

User modelling is complex, perhaps one of the more complex applications yet encountered in applied computer science and artificial intelligence research. It is not possible to provide complete approaches in a single research effort, and solutions are neither singular nor simple. This does not mean the effort is not important nor should it be abandoned. Personalized computer systems that adapt to our needs, are able to give and explain meaningful advice, and can interpret our actions are a consistent image in science fiction and futuristic scenarios studied by serious researchers [Skulley 88]. User modelling is one of the important enabling technologies needed to reach that goal; but arriving at a common, useful theory will require multiple efforts and the synthesis of results in order to fully understand the cognitive and computational issues involved.

References

[Anderson, Conrad, Corbett 89]
J.R. Anderson, F.G. Conrad, A.T. Corbett, *Skill Acquisition and the LISP Tutor*, Cognitive Science, Vol. 13, 1989, pp. 467-505.

[Atwood et al. 90]
M.E. Atwood, W.D. Gray, B. Burns, A.I. Morch, B. Radlinski, *Cooperative Learning and Cooperative Problem Solving: The Case of Grace*, Working Notes, 1990 AAAI Spring Symposium on Knowledge-Based Human-Computer Communication, AAAI, Menlo Park, CA, 1990, pp. 6-10.

[Bloom 84]
B.S. Bloom, *The Search for Methods of Group Instruction as Effective as One-to-One Tutoring*, Educational Leadership, May 1984, pp. 4-17.

[Brech, Jones 88]
B. Brecht, M. Jones, *Student Models: the Genetic Graph Approach*, International Journal of Man-Machine Studies, Vol. 28, 1988, pp. 483-503.

[Britton, Black 85]
Bruce K. Britton, John B. Black (eds.), *Understanding Expository Text*, Lawrence Erlbaum Associates, London, 1985.

[Brown, Burton 86]
J.S. Brown, R.R. Burton, *Reactive Learning Environments for Teaching Electronic Troubleshooting*, in W.B. Rouse (ed.), *Advances in Man-Machine Systems Reasearch, Vol 3*, JAI Press, Inc, Greenwich, CT, 1986.

[Brown, Burton, Kleer 82]
J.S. Brown, R.R. Burton, J. de Kleer, *Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I, II and III*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 227-281, ch. 11.

[Burton, Brown 82]
R.R. Burton, J.S. Brown, *An Investigation of Computer Coaching for Informal Learning Activities*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 79-98, ch. 4.

[Burton, Brown, Fischer 84]
R.R. Burton, J.S. Brown, G. Fischer, *Analysis of Skiing as a Success Model of Instruction: Manipulating the Learning Environment to Enhance Skill Acquisition*, in B. Rogoff, J. Lave (eds.), *Everyday Cognition: Its Development in Social Context*, Harvard University Press, Cambridge, MA - London, 1984, pp. 139-150.

[Carbonell 70]
J.R. Carbonell, *AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction*, IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, December 1970.

[Carroll, Carrithers 84]
J.M. Carroll, C. Carrithers, *Training Wheels in a User Interface*, Communications of the ACM, Vol. 27, No. 8, August 1984, pp. 800-806.

[Carroll, McKendree 87]
J.M. Carroll, J. McKendree, *Interface Design Issues for Advice-Giving Expert Systems*, Communications of the ACM, Vol. 30, No. 1, January 1987, pp. 14-31.

[Chandrasekaran, Tanner, Josephson 88]
B. Chandrasekaran, C. Tanner, J.R. Josephson, *Explanation: The Role of Concept Strategies and Deep Models*, in J.A. Hendler (ed.), *Expert Systems: The User Interface*, Ablex Publishing Corp, Norwood, NJ, 1988.

[Chandrasekaran, Tanner, Josephson 89]
B. Chandrasekaran, C. Tanner, J.R. Josephson, *Explaining Control Strategies in Problem Solving*, IEEE Expert, Vol. 4, No. 1, Spring 1989, pp. 9-23.

[Clancey 84]
W. Clancey, *Use of MYCIN's Rules for Tutoring*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 464-489, ch. 26.

[Clancey 87]
W.J. Clancey, *Knowledge-Based Tutoring: The Guidon Program*, MIT Press, Cambridge, MA, 1987.

[Dews 89]
S. Dews, *Developing an ITS in a Corporate Setting*, Proceedings of the 33rd Annual Meeting of the Human Factors Society (Denver, CO), 1989, pp. 1339-1342.

[Dijk, Kintsch 83]
T.A. van Dijk, W. Kintsch, *Strategies of Discourse Comprehension*, Academic Press, New York, 1983.

[Duchastel 88]
P.C. Duchastel, *Models for AI in Education and Training*, Artificial Intelligence Tools in Education: Proceedings of the IFIP TC3 Working Conference, IFIP, 1988, pp. 17-28.

[Fischer 84]
G. Fischer, *Formen und Funktionen von Modellen in der Mensch-Computer Kommunikation*, in H. Schauer, M.J. Tauber (eds.), *Psychologie der Computerbenutzung*, R. Oldenbourg Verlag, Wien - Muenchen, Schriftenreihe der Oesterreichischen Computer Gesellschaft, Vol. 22, 1984, pp. 328-343.

[Fischer 87]
G. Fischer, *Learning on Demand: Ways to Master Systems Incrementally*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1987.

[Fischer 88a]
G. Fischer, *Enhancing Incremental Learning Processes with Knowledge-Based Systems*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, New York, 1988, pp. 138-163, ch. 7.

[Fischer 88b]
G. Fischer, *Cooperative Problem Solving Systems*, Proceedings of the 1st Simposium Internacional de Inteligencia Artificial (Monterrey, Mexico), October 1988, pp. 127-132.

[Fischer 90]
G. Fischer, *Communications Requirements for Cooperative Problem Solving Systems*, The International Journal of Information Systems (Special Issue on Knowledge Engineering), Vol. 15, No. 1, 1990, pp. 21-36.

[Fischer et al. 88]
G. Fischer, S.A. Weyer, W.P. Jones, A.C. Kay, W. Kintsch, R.H. Trigg, *A Critical Assessment of Hypertext Systems*, Human Factors in Computing Systems, CHI'88 Conference Proceedings (Washington, DC), ACM, New York, May 1988, pp. 223-227.

[Fischer et al. 90]
G. Fischer, A.C. Lemke, T. Mastaglio, A. Morch, *Using Critics to Empower Users*, Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA), ACM, New York, April 1990, pp. 337-347.

[Fischer, Lemke 88]
G. Fischer, A.C. Lemke, *Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication*, Human-Computer Interaction, Vol. 3, No. 3, 1988, pp. 179-222.

[Fischer, Lemke, Mastaglio, Morch 90]
G. Fischer, A. Lemke, T. Mastaglio, A. Morch, *Critics: An Emerging Approach to Knowledge-Based Human Computer Interaction*, International Journal of Man-Machine Studies, 1990, to be published.

[Fischer, Lemke, Schwab 84]
G. Fischer, A.C. Lemke, T. Schwab, *Active Help Systems*, Readings on Cognitive Ergonomics - Mind and Computers, Proceedings of the 2nd European Conference (Gmunden, Austria), G.C. van der Veer, M.J. Tauber, T.R.G. Green, P. Gorny (eds.), Springer-Verlag, Berlin - Heidelberg - New York, September 1984, pp. 116-131.

[Fischer, Mastaglio 89]
G. Fischer, T. Mastaglio, *Computer-Based Critics*, Proceedings of the 22nd Annual Hawaii Conference on System Sciences, Vol. III: Decision Support and Knowledge Based Systems Track, IEEE Computer Society, January 1989, pp. 427-436.

[Fischer, Mastaglio, Reeves, Rieman 90]
G. Fischer, T. Mastaglio, B. Reeves, J. Rieman, *Minimalist Explanations in Knowledge-Based Systems*, Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr (ed.), IEEE Computer Society, 1990, pp. 309-317.

[Fischer, Mastaglio, Rieman 89]
G. Fischer, T. Mastaglio, J. Rieman, *User Modeling in Critics Based on a Study of Human Experts*, Proceedings of the Fourth Annual Rocky Mountain Conference on Artificial Intelligence, RMSAI, Denver, CO, June 1989, pp. 217-225.

[Flesch 49]
R. Flesch, *The Art of Readable Writing*, Harper & Brothers, New York, 1949.

[Fox 88]
B.A. Fox, *Robust learning environments -- the issue of canned text*, Technical Report, Institute of Cognitive Science, University of Colorado, Boulder, Colorado, 1988.

[Frank, Lynn, Mastaglio 87]
J. Frank, P. Lynn, T. Mastaglio, *Using A Critic Methodology as a Computer-aided Learning Paradigm: extending the concepts*, 1987, Final Project Report for CS659 - Fall Term 1987.

[Gentner, Stevens 83]
D. Gentner, A.L. Stevens (eds.), *Mental Models*, Lawrence Erlbaum Associates, Hillsdale, NJ, Cognitive Science Series, 1983.

[Goldstein 82]
I.P. Goldstein, *The Genetic Graph: A Representation for the Evolution of Procedural Knowledge*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 51-77, ch. 3.

[Hansen, Hass 88]
W.J. Hansen, C. Haas, *Reading and Writing with Computers: A Frameworks for Explaining Differences in Performance*, Communication of the ACM, Vol. 231, No. 9, September 1988, pp. 1080-1089.

[Hefley 90]
W. Hefley, *Architectures for Adaptable Human-Machine Interfaces*, in Karwowski, Rahimi (eds.), *Ergonomics of Advanced Manufacturing and Hybrid Automation Systems II*, Elsevier, N.Y., 1990, forthcoming.

[Hempel 65]
C.G. Hempel, *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*, The Free Press, New York,, 1965.

[Johnson, Soloway 84]
W.L. Johnson, E. Soloway, *PROUST: Knowledge-Based Program Understanding*, Proceedings of the 7th International Conference on Software Engineering (Orlando, FL), IEEE Computer Society, Los Angeles, CA, March 1984, pp. 369-380.

[Kass 87a]
R. Kass, *Implicit Acquisition of User Models in Cooperative Advisory Systems*, Technical Report MIS-CIS-87-05, LINC LAB 49, University of Pennsylvania, 1987.

[Kass 87b]
R. Kass, *Modelling User Beliefs for Good Explanations*, Technical Report MIS-CIS-87-77, LINC LAB 82, University of Pennsylvania, 1987.

[Kass 88]
R. Kass, *Acquiring a Model of the User's Belief from a Cooperative Advisory Dialog*, Unpublished Ph.D. Dissertation, University of Pennsylvania, 1988.

[Kass, Finin 87]
R. Kass, T. Finin, *Rules for the Implicit Acquisition of Knowledge about the User*, 6th National Conference on Artificial Intelligence, AAAI, 1987, pp. 295-300.

[Kass, Finin 88a]
R. Kass, T. Finin, *A General User Modelling Facility*, CHI '88 Conference Proceedings, Human Factors in Computing Systems, ACM, 1988, pp. 145-150.

[Kass, Finin 88b]
R. Kass, T. Finin, *The Need for User Models in Generating Expert System Explanations*, International Journal of Expert Systems, Vol. 4, 1988, pp. 345-375.

[Kennedy etal. 88]
A. Kennedy, A. Wildes, L. Elder, W.S. Murray, *Dialogue with Machines*, Cognition, Vol. 30, 1988, pp. 37-72.

[Kintsch 89]
W. Kintsch, *The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension*, in R. Dietrich, C.F. Graumann (eds.), *Language Processing in Social Context*, North Holland, Amsterdam, 1989, pp. 185-209, also published as Technical Report No. 152, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[Kobsa, Wahlster 89]
A. Kobsa, W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag, New York, 1989.

[Lemke 89]
A.C. Lemke, *Design Environments for High-Functionality Computer Systems*, Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado, July 1989.

[Manheim, Srivastava, Vlahos, Hsu, Jones 90]
M.L. Manheim, S. Srivastava, N. Vlahos, J. Hsu, P. Jones, *A Symbiotic DSS for Production Planning and Scheduling: Issues and Approaches*, Proceedings of the 23rd Annual Hawaii International Conference on System Sciences, Vol III, J.F. Nunamaker, Jr. (ed.), Jan 1990, pp. 383-390.

[Mastaglio 89]
T. Mastaglio, *Computer-based Critiquing: A Foundation for Learning Environments*, Proceedings TITE '89, 1989 Conference on Technology and Innovations in Training and Education, March 6-9, 1989, Atlanta, GA, Linda Wiekhorst (ed.), 1989, pp. 125-136.

[Mastaglio 90a]
T. Mastaglio, *Paradigms for Intelligent Learning Environments: Tutoring, Coaching and Critiquing*, Proceedings TITE '90, 1990 Conference on Technology and Innovations in Training and Education, March 12-16, 1990, Colorado Springs, CO, 1990, pp. 190-204.

[Mastaglio 90b]
T. Mastaglio, *User Modelling in Computer-Based Critics*, Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr (ed.), IEEE Computer Society, 1990, pp. 403-412.

[Mastaglio, Reeves 90]
T. Mastaglio, B. Reeves, *Explanations in Cooperative Problem Solving Systems*, Proceedings 12th Annual Conference of the Cognitive Science Society, July 25-28, 1990, 1990, pp. 301-308.

[Mili, Manheim 88]
F. Mili, M.L. Manheim, *And What Did Your DSS Have to Say About That: Intoduction to the DSS Minitrack on Active and Symbiotic Systems*, Proceedings of the 21st Hawaii International Conference on System Sciences, Jan 1988, pp. 1-2.

[Miller 79]
M.L. Miller, *A Structured Planning and Debugging Environment for Elementary Programming*, in D.H. Sleeman, J.S. Brown (eds.), *International Journal of Man-Machine Studies*, Academic Press, 1979, pp. 79-95.

[Moore 87]
J. Moore, *Explanations in Expert Systems*, Technical Report, USC/Information Sciences Institute, 9 December 1987.

[Moore 89]
J. Moore, *Responding to 'HUH': Answering Vaugely Articulated Follow-up Questions*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 91-96.

[Murray 88]
D. Murray, *A Survey of User Cognitive Modelling*, Technical Report NPL Report 92/87, DITC, National Physical Laboratory, Teddinton, Middlesex TW11 0LW, UK, 1988.

[Neches, Swartout, Moore 85]
R. Neches, W.R. Swartout, J.D. Moore, *Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1337-1351.

[Norman 82]
D.A. Norman, *Five Papers on Human-Machine Interaction*, CHIP Report 112, University of California, San Diego, May 1982.

[Paris 87]
C.L. Paris, *The Use of Explicit User Models in Text Generation: Tailoring to a User's Level of Expertise*, Unpublished Ph.D. Dissertation, Columbia University, 1987.

[Paris 89]
C.L. Paris, *The Use of Explicit User Models in a Generation System for Tailoring Answer to a User's Level of Expertise*, in A. Kobsa, W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag, New York, 1989, pp. 200-232.

[Polson, Richardson 88]
M.C. Polson, J.J. Richardson (eds.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[Psotka, Massey, Mutter 88a]
J. Psotka, L.D. Massey, S.A. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[Psotka, Massey, Mutter 88b]
J. Psotka, L.D. Massey, S. Mutter, *Intelligent Instructional Design*, in J. Psotka, L.D. Massey, S. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ , 1988, pp. 113-118.

[Rich 79]
E. Rich, *Building and Exploiting User Models*, Unpublished Ph.D. Dissertation, Carnegie-Mellon University, 1979.

[Rich, Waters 88]
C.H. Rich, R.C. Waters, *Automatic Programming: Myths and Prospects*, Computer, Vol. 21, No. 8, August 1988, pp. 40-51.

[Rich, Waters 90]
C. Rich, R.C. Waters, *The Pogrammer's Apprentice*, ACM Press, New York, 1990.

[Schank 86]
R.G. Schank, *Explanation Patterns: Understanding Mechanically and Creatively*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[Scott, Clancey, Davis, Shortliffe 84]
A.C. Scott, W.J. Clancey, R. Davis, E.H. Shortliffe, *Methods for Generating Explanations*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 338-362, ch. 18.

[Shortliffe 76]
E.H. Shortliffe, *Computer-Based Medical Consultations: MYCIN*, Elsevier, New York - Amsterdam, Artificial Intelligence Series, Vol. 2, 1976.

[Skulley 88]
J. Skulley, *The Relationship Between Business and Higher Education: A Perspective on the 21st Century*, CACM, Vol. 32, No. 9, September 1988, pp. 1056-1061.

[Sleeman 84]
D.H. Sleeman, *UMFE: A User Modeling Front End Subsystem*, Working Paper HPP-84-12, Heuristic Programming Project, Department of Computer Science, Stanford University, April 1984.

[Sleeman, Brown 82]
D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, Computer and People Series, 1982.

[Steele 84]
G.L. Steele, *Common LISP: The Language*, Digital Press, Burlington, MA, 1984.

[Strunk, White 57]
W. Strunk, E.B. White, *The Elements of Style*, Harcourt-Brace, New York, 1957.

[Suchman 87]
L.A. Suchman, *Plans and Situated Actions*, Cambridge University Press, New York, 1987.

[Swartout 83]
W.R. Swartout, *XPLAIN: A System for Creating and Explaining Expert Consulting Programs*, ISI Reprint Series ISI/RS-83-4, Information Sciences Institute, University of Southern California, Marina del Rey, CA, July 1983.

[Teach, Shortliffe 84]
R.L. Teach, E.H. Shortliffe, *An Analysis of Physicians' Attitudes*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 635-652, ch. 34.

[VanLehn 88]
K. VanLehn, *Toward a Theory of Impasse-Driven Learning*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, New York, 1988, pp. 19-41, ch. 2.

[Wahlster, Kobsa 88]
W. Wahlster, A. Kobsa, *User Models in Dialog Systems*, Technical Report 28, Universitaet des Saarlandes, FB 10 Informatik IV, Sonderforschungsbereich 314, Saarbruecken, FRG, 1988.

[Wallis, Shortliffe 84]
J.W. Wallis, E.H. Shortliffe, *Customized Explanations Using Causal Knowledge*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 371-388, ch. 20".

[Weiss 88]
E.H. Weiss, *Breaking the Grip of User Manuals*, Asterisk -- Journal of ACM SIGDOC, Vol. 14, Summer 1988, pp. 4-11.

[Wenger 87]
E. Wenger, *Artificial Intelligence and Tutoring Systems*, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Wiener 80]
J.L. Wiener, *BLAH, A System Which Explains its Reasoning*, Artificial Intelligence, Vol. 15, 1980, pp. 19-48.

[Wilensky 84]
R. Wilensky, *LISPcraft*, W.W. Norton & Company, New York - London, 1984.

[Wilkins, Clancey, Buchanan 88]
D.C. Wilkins, W.J. Clancey, B.G. Buchanan, *Using and Evaluating Differential Modelling in Intelligent Tutoring and Apprentice Learning Systems*, in J. Psotka, L.D. Massey, S. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ , 1988, pp. 257-277.

[Winograd, Flores 86]
T. Winograd, F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ, 1986.

[Winston, Horn 81]
P.H. Winston, B.K.P. Horn, *LISP*, Addison-Wesley Publishing Company, Reading, MA, 1981.

INDEX