A conceptual framework for knowledge-based critic systems

Gerhard Fischer and Thomas Mastaglio *

Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder, CO 80309, USA

The critiquing paradigm is one approach to instantiating the concept of intelligent support systems. Knowledge-based systems that use the critiquing approach can support numerous application domains, including: programming, design and decision making. Critiquing is an alternative to expert systems that can support cooperative problem solving and aid user learning in the application domain. As a result of empirical studies we identified the requirements for critic systems. We have developed several knowledge-based critics to instantiate these ideas and used them to identify new issues and theory. Our systems have been revised to include approaches to addressing these issues. As a result of these implementations we have developed a general architecture for knowledge-based critics that fully support cooperative problem solving. We describe the current extension of one of our systems designed to incorporate these findings.

Keywords: Cooperative problem solving, Critics, Design environments, User models, Explanation, LISP-CRITIC, FRAMER, JANUS.

* *Acknowledgments*: Many people have contributed to the development of LISP-CRITIC over the last few years. The authors would like to thank especially; Heinz-Dieter Boecker, who developed many of the original ideas, the original set of rules and the rule interpreter; Andreas Lemke developed FRAMER; Helga Nieper-Lemke developed KAESTLE; Christopher Morel, Bart Burns, and Catherine Cormack contributed to version 1 of LISP-CRITIC; Anders Morch developed JANUS; John Rieman, Paul Johl, and Patrick Lynn worked on version 2 of LISP-CRITIC; Hal Eden and Brent Reeves for recent work on LISP-CRITIC. This research is partially supported by grants from: NYNEX, Software Research Associates (SRA), and the Colorado Institute of Artrticial Intelligence (CIAI). The second author was supported during graduate studies at the University of Colorado by the U.S. Army.

1. Introduction

Our goal is to establish the conceptual foundations for using the computational power that is or will be available on computer systems. We believe that artificial intelligence technologies can improve productivity by addressing, rather than ignoring, human needs and potential. In the spirit of Einstein's remark "*My pencil is cleverer than I*" we are building systems that augment and amplify human intelligence. Winograd and Flores [63] argue that the development of tools for conversation, the computer serving as a structured dynamic

Gerhard Fischer is professor of computer science and a member of the Institute of Cognitive Science at the University of Colorado at Boulder. He directs the university's "Knowledge-Based Systems and Human-Computer Communication" research group. Before joining the University of Colorado, he directed a similar group at the University of Stuttgart, W-Germany. Research interest include artificial intelligence, human-computer communication, cognitive science and software engineering; he is especially interested in bringing these research disciplines together to build cooperative problem solving systems. His research group has constructed a rich variety of tools and a large number of application systems to test the theories and methods guiding this research. Dr. Fischer received a PhD in computer science from the University of Hamburg. He can be contacted at Computer Science Department, University of Colorado, CO 80309-0430; CSnet: gerhard@boulder.colorado.edu.

Thomas W. Mastaglio is an Army Officer assigned to the U.S. Army Training and Doctrine Command, Fort Monroe, Virginia. He is responsible for integrating emerging technologies into future training and support systems. He is native of Wisconsin and received his B.S. and Commission from the U.S. Military Academy in 1969. Lieutenant Colonel Mastaglio holds Master of Science and Doctor of Philosophy Degrees in Computer Science from the University of Colorado. His research interests include applying artificial intelligence techniques to support computer based training, and decision making. His specific research involves user modelling techniques in critiquing systems. He can be contacted at Headquarters US Army TRADOC, ATTN: ATTG-U, Fort Monroe, VA 23651, CSnet: mastaglt%mon1@leavemh.army.mil.

medium for conversation in systematic domains, is a more realistic and relevant way of exploiting information and communication technologies than is the most widely perceived goal of artificial intelligence, "to understand and to build autonomous, intelligent, thinking machines" [57]. We argue that intermediate approaches, ones that use knowledge-based techniques to assist users in their application domains, need to be investigated.

"Intelligent support systems" are used as a generic name for systems that augment human capabilities. The major application domain of our intelligent support systems have been high functionality computer systems, such as UNIX or LISP machines which contain tens of thousands of objects and tools. Developing intelligent support systems has as its goal making usable the total space of functionality available in computational environments rather than diluting functionality or orienting the user on only a subset of the system's capabilities.

Intelligent support systems should facilitate access, application of knowledge, and user learning. We have constructed a number of different intelligent support systems: documentation systems [24], active and passive help systems [17], design environments [15;34], and critics [11;22;18]. All of these systems have two things in common: they are knowledge-based and they use innovative techniques in humancomputer communication. In this paper we focus on knowledge-based critics.

The ultimate objective of our efforts is to evolve intelligent support systems to cooperative problem solving systems. Cooperative problem solving characterizes situations in which intelligent agents work together to produce a design, plan, product or decision. Our work has focused on dyadic situations with a human problem solver and knowledge-based computer system.

In this article we describe the conceptual framework for knowledge-based critics and some of the general principles we have learned from building critics. Section 2 presents the general framework for critics and section 3 presents their specific requirements. Section 4 describes prototypical critic systems that we have constructed: LISP-CRITIC, which critiques LISP programs, and two design environments: FRAMER for interface construction and JANUS kitchen design. Section 5 elaborates further the notion of cooperative problem solving and explains how the design which has evolved for the latest version of LISP-CRITIC supports this idea.

2. A characterization of the critic paradigm

The critiquing approach tries to make use of available computer knowledge bases to aid users in their own work. A colleague can often, with little time and effort, provide the missing link that we, just on our own, cannot find after hours or days of work. This happens in many problem solving domains, for example, computer programming and other design tasks.

Artificial intelligence research was initially directed towards creating autonomous intelligent agents. However, in many situations it is desirable to keep a "human in the loop". There are several reasons for this. Often the intelligent system does not have the knowledge required to cover the complete problem domain and interaction with humans is inevitable. On the other hand, humans should not be deskilled by their computer systems, demoted to mere suppliers of data. Computer systems should function in a cooperative mode where the human and the machine collaborate in working towards a common goal. Instead of autonomous expert systems we want to develop intelligent support systems where the abilities of the computer and the human are synergized to form a "joint human–machine cognitive system" [65].

One source of information available to an intelligent support system is the human's interaction with the system. This interaction could be viewed as a crude dialog and it along with the partial products that are created during this interaction can be used by the computer as insight into what the human is doing. If the human's primary work environment is a computer system, then these actions are accessible to the system as sequences of commands, menu selections, mouse operations, and the like. Critic systems exploit those sources available to the computer to facilitate cooperative interaction with a user during problem solving.

Critics analyze a product produced by the user and provide suggestions as to how the user can improve that product, if the user so desires, in some domains the system may be able to incorporate those suggestions directly in a revised version of the product. The process is cyclical and continues until the user is satisfied with the solution.

The roles played by the computer and the human in this process are interactive and interdependent, they are depicted in fig. 1.

It is probably instructive to clarify the distinction between critics and constraints. A significant aspect of critiquing is that the user remains in control and is free to accept or reject advice from the critic. Constraints are often "hard coded" into the working environment of systems or enforced on the user by some other system process (e.g., My file name extension in MS/DOS cannot be more than 3 characters); they are narrowly focused criteria that must be adhered to in order for something to function properly. Critiquing focuses on improving the functionality of a product that is already usable. The expertise that critics possess is based on soft constraints.

Critics also are a method for using knowledge-based approaches to support ill-structured problem domains. Expert systems have generally attacked problems in well defined and tightly constrained problem spaces. Attempts to develop autonomous expert systems for ill structured domains have not been as successful. The critiquing paradigm allows the knowledge-based System to contribute whatever knowledge it has to assist users with their work in these "fuzzier" problem domains.

2.1. Intelligent support systems

Empirical investigations [10;17] have shown that on the average only a small fraction of the functionality of complex systems such as UNIX, EMACS and LISP is used. Consequently it will be of little use to equip modern computer systems with more and more computational power and functionality, unless we can help the user take advantage of them. The "intelligence" of a complex computer system must contribute to its ease of use and provide for effective communication. Intelligent and knowledgeable human communicators, for example, good teachers, have substantial knowledge about how to explain their expertise to others.

It is not sufficient for intelligent support systems to just solve a problem or provide information. Users must be able to understand these systems and question their advice. One of our assumptions is that learners and practitioners will not ask a computer program for advice if they have no way to examine the program's expertise.

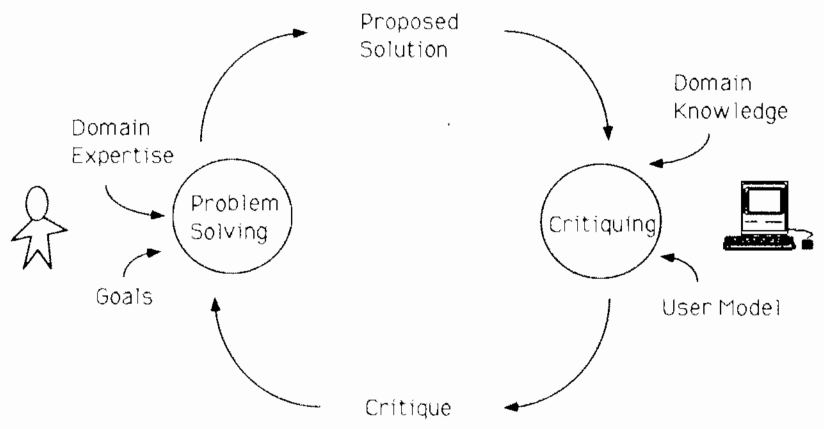


Fig. 1. The critiquing process. This diagram shows the critiquing process. It is significant that the human user and the computer each possess domain knowledge that is brought to bear on the problem. The user applies his or her domain expertise during problem solving to generate a proposed solution that will potentially accomplish his or her goals. The knowledge-based critic applies domain knowledge to critique that product. This process continues until the user is satisfied with the solution produced.

Users must be able to access the system's knowledge and reasoning processes; domain knowledge has to be explainable [59]. A system that possesses domain knowledge has to be capable of sharing that knowledge with the user at the application level as well as at the conceptual level.

2.2. Cooperative problem solving in critic systems

One model frequently used in human-computer systems (e.g., MYCIN [4]) is the consultation model. From an engineering point of view, it has the advantage of being clear and simple: the program controls the dialog, much as a human consultant does, by asking for specific items of data about the problem at hand. It precludes the user volunteering what he or she might think is relevant data. The program is viewed as an "all-knowing expert", and the user is left in the undesirable position of asking a machine for help.

A more appealing model, one that more closely approximates human to human collaboration, is the cooperative problem solving approach [13]. Problem solving effectiveness is often enhanced by cooperation – traditionally cooperation among people, or more recently, cooperation between a human and a computer. The emphasis of our work is on creating computer systems to facilitate the cooperation between a human and a knowledge-based computer. Examination of these systems provides evidence that learning and effective problem solving can be improved through the use of cooperative systems. It also indicates the need for a richer theory of problem solving, which analyzes the function of shared representations [15], mixed-initiative dialogues [25], and the management of trouble [36].

Because there is an asymmetry between humans and computers, the design of cooperative problem solving systems should not only simulate human to human cooperation, but develop engineering alternatives. For example, natural language may not always be the preferred mode of communication. We have studied situations in which human to human cooperation naturally occurs between customers and sales agents. In one of these situations we recorded and analyzed problem solving dialogs that took place in a very large hardware store [45]. This store – McGuckin hardware – is reputed for its large inventory and the ability of its sales agents to aid customers in finding the items they need to accomplish a task, i.e., to help customer's solve their problems. The results of this study have provided insights into the processes that are important for cooperative problem solving.

Depending on the task and the knowledge which the user has, different role distributions (e.g., tutors [1], suggestors [15], advisors [8], or critics [18]) should be supported. In our work, we have concentrated on the critic paradigm which will be further discussed in the next section. The critiquing model supports cooperative problem solving. When two agents (e.g., a learner and a critic) communicate, much more goes on than just a request for factual information. Learners may not be able to articulate their questions without the help of the critic; the advice given by the critic may not be understood, and/or learners require explanations. Each communication partner may hypothesize that the other partner misunderstood him/her, or the critics might give advice for which they were not explicitly asked.

Beyond the domains which we have studied (see section 4), decision support systems can also be viewed as an instance of the class of cooperative problem solving systems. They should aid users by providing context sensitive advice about how to improve the users' decision making or see their problem from a different viewpoint. Expert systems usually are based on a closed world assumption, and are inadequate replacements for human-oriented decision support systems that have to be flexible and adaptive [60]. The cooperative problem solving approach using a critiquing methodology seems to be a more promising approach for integrated decision support system.

2.3. Support for contextual learning

Critiquing can support the contextualization of learning [37] by supporting user-centered learning, incremental learning, and learning on demand.

User-centered learning. User-centered learning allows individuals to follow different learning paths. Forcing the same intellectual style on every individual is possibly much more damaging than forcing right-handedness upon a left-hander. To support user-entered learning processes, computational environments have to adapt to individual needs and learning styles. Giving users control over their learning and work requires them to initiate actions and set their own goals. Critics require individualized representations of domain knowledge to support explanations. They can use them to present explanations which relate new concepts to knowledge previously held by a specific user.

Incremental learning. Not even experts can completely master complex, high-functionality systems. Support for incremental learning is required. Incremental learning eliminates suboptimal behavior (thereby increasing efficiency), enlarges possibilities (thereby increasing functionality), supports learning on demand by presentation of new information when it is relevant, uses models of the user to make systems more responsive to the needs of individuals, and tailors explanations to the user's conceptualization of the task.

Learning on Demand. The major justification for learning on demand is that education is a distributed, lifelong process of learning material as it is needed. Learning on demand has been successful in human societies or organizations when learners are afforded the luxury of a personal coach or critic. Aided by a human coach or critic, learners can articulate their problems in an infinite variety of ways. Computerbased support systems should be designed to conform to this metaphor.

On a broad scale, learning on demand is neither practical nor economical without computers. Learning on demand should include "learning to learn," showing the user how to locate and utilize information resources. It should not be restricted just to learning procedures but should help to restructure the user's conceptual model of the domain. It should not only provide access to factual information but also assist the user in understanding when that knowledge can be applied.

Learning on demand is a guided discovery approach to learning (see section 4.3 for a description of one of our systems that supports this idea.) It is initiated when the user wants to do something, not learn about everything. Learning on demand affords that learning occurs because knowledge is actively used rather than passively perceived, at least one condition under which knowledge can be applied is learned and it can make a crucial difference in motivating learning.

Learning on demand can be differentiated according to whether the user or the system initiates the demand:

- *Demands Originating with the User.* The demand to learn more can originate with the user. It can be triggered by a discrepancy between an intended product and the actual product produced. Experimentation with a system may turn up interesting phenomena that users find worth exploring further. The user's mental model can serve as a driving force towards learning more. Users' "feel" that there must be a better way of doing things. Adequate tools to support learning on demand are crucially important in making users willing to embark on an effort to increase their knowledge.
- *Suggestions from the Coach or the Critic.* The demand to learn cannot originate with users when they are unaware that additional functionality exists in the system. The system has to take the initiative. To avoid the problem of these systems becoming too intrusive, a metric is necessary for judging the adequacy of a user's action. Except for narrow problem domains (e.g., simple games [6]), optimal behavior cannot be uniquely defined. Therefore, the underlying metric that determines the behavior of a coach or a critic should not be a fixed entity but a structure that users can inspect and modify according to their view of the world. This increases the user's control over interaction with the system. Adequate communication structures must exist to make this task manageable.

A major challenge for this class of systems is to keep them quiet most of the time. A coach or a critic must be capable of diagnosing the cause of a student's misunderstanding and then judiciously deciding, on its own, when to interrupt and what to say. Interrupting too often can destroy motivation, but too few interruptions results in learning experiences being missed or floundering on the part of the user.

2.4. Related work

The critiquing approach was used in research efforts on medical systems [40;41;33;48]. These systems use domain knowledge to help physicians perform diagnoses or develop patient treatment plans. Techniques from expert systems research were modified after researchers recognized the need to assist physicians directly in their work, leaving them in control rather than attempting to replace them with an autonomous system.

Critics have also been develop to support circuit design [31;56], to teach decision making to managerial personnel [51], and to improve the performance of decision makers, not through training, but in the context of their actual work [39]. Other widely differing applications include teaching fundamentals of circuit design [28], curriculum development [64], and improving written compositions [27].

Our research and system development efforts have come from a perspective of human-computer interaction. We ask how knowledge-based approaches can improve collaboration between a computer and a user. We have built on research in advice giving systems [8], explanation approaches [58;43;52;46;42], and user modelling [49;7;54;30].

3. Requirements for critic systems

Design requirements for computer-based critics should be based on empirical studies. As we have studied human critics, it became obvious that knowledge is the most important feature of a good critic. This knowledge must be available in a form useful for constructing critiques and for aiding user understanding of those critiques.

3.1. Empirical studies

Cognitive scientists have studied human-to-human dyadic relationships. These studies emphasized psychological [9] and linguistic [26] aspects of dyadic human cooperative efforts. In our own work, we have investigated the problems users encounter in dealing with high functionality computer systems:

- Users do not know about the existence of tools and therefore are not able to ask for them; passive help systems are of little use in such situations [17].
- Users do not know how to access tools; retrievability is a big problem in information-rich societies and in complex, high-functionality systems [23].
- Users do not know when to use these tools; that is, they do not know the applicability conditions under which a piece of knowledge can be used successfully [14].
- Users do not understand the results that tools produce. Finding the information is in many cases not the end but the beginning of difficulties [3].
- Users cannot combine, adapt, and modify a tool to their specific needs; reuse and redesign have to be supported [12].
- Users encounter difficulty mapping their situation model into the available resources represented in terms of the system model. They need either better techniques for accomplishing this mapping or knowledgeable agents to assist them [45].

A consequence of these problems is that many systems are underused. We are convinced that what is needed is not more information but new ways to structure and present it. Presenting information entails producing explanations appropriate for each individual user's expertise which in turn requires our systems to acquire and maintain individual models of users. Another empirical study (based on thinking-aloud protocols from experts) [20] investigated how such a model of the expertise of another user is acquired by a domain expert. The analysis showed that human experts look for certain cues that trigger inferences about the user. Our systems are based on such an approach, but they are not able to use all the evidence available in human to human interaction. It is, however, our goal to take advantage of that information which the computer can access and use.

The design of our critic systems has been influenced by these empirical studies. Our approach is based on two assumptions: that cooperative work is a powerful approach to both improving problem solving and learning, and that users need to be encouraged to explore. Decision support systems can benefit from our efforts to create such environments because they share many of the same goals as well as limitations that we have encountered in our studies and system building efforts.

3.2. Knowledge-based architectures

Knowledge-based systems are one promising approach to equipping machines with some human communication capabilities. Based on an analysis of human communication, we have developed the model shown in fig. 2, and tried to instantiate this general architecture in a variety of systems.

The system architecture in fig. 2 contains two major improvements over traditional approaches:

- The explicit communication channel is widened (incorporating the use of windows, menus, pointing devices, inter-referential input/output, etc.). By representing systems in the world model [29] (for examples see figs. 9 and 11), Users can inspect and manipulate these models directly.
- Information can be exchanged over the implicit communication channel relying on shared knowledge structures. This eliminates the need to specify operations in detail and reduces the conceptual distance between the domain itself and the way the users communicate about it.

Knowledge-based communication requires that users and systems have knowledge in the following domains (see fig. 2):

Knowledge about the problem domain. Expertise cannot exist without domain knowledge. Intelligent behavior builds upon large amounts of knowledge about specific domains. This knowledge constrains the number of possible actions and describes reasonable goals and operations. Most computer users are not interested in computers per se but want to use the computer to solve problems and accomplish tasks. To shape the computer into a truly usable and useful medium for them, we have to make it invisible and let them work directly on their problems and their tasks; that is, we must support human problem-domain communication [15]. A representation of domain knowledge is needed that can be used to explain the underlying concepts of the domain and model the user. For the applicable form of domain knowledge we have used rule-based systems because they support the incremental accumulation of domain knowledge and are efficient. An ideal system would be able to generate the applicable form of the domain knowledge, e.g., rules, from the more abstract conceptual representation [58]. Thus far we have not achieved that goal but rather have incorporated dual representations into our systems.

Knowledge about communication processes. The information structures that control communication should be made explicit. This will allow users to refer to them (e.g., history lists of commands, bookmarks of places visited before). Exploratory approaches can be encouraged by supporting UNDO and REDO commands.

Knowledge about the communication partner. The

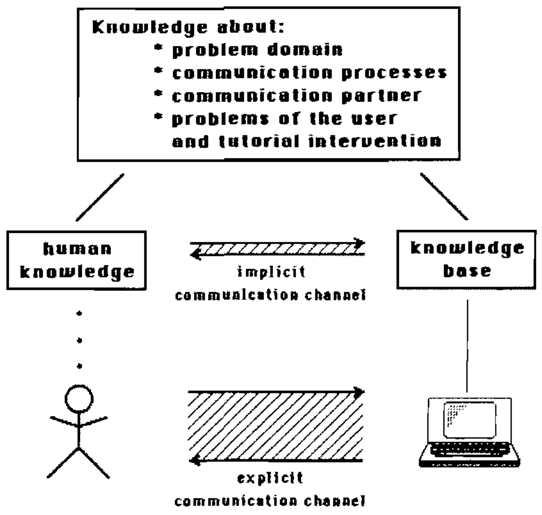

Knowledge about:
* problem domain
* communication processes
* communication partner
* problems of the user and tutorial intervention
human knowledge
implicit communication channel
knowledge base
explicit communication channel

Fig. 2. Architecture for knowledge-based human-computer communication.

user of a system does not exist; there are many different kinds of users, and the requirements of an individual user change with experience. Systems will be unable to interact with users intelligently unless they have some means of finding out what the user really knows; they must be able to infer the state of the user's knowledge. To support incremental learning and learning on demand, systems must possess knowledge about a specific user, information about the user's conceptual understanding of a system, the set of tasks for which the user uses the system, the user's way of accomplishing domain-specific tasks, the pieces of advice given and whether they were remembered and accepted, and the situations in which the user asked for help. Our approach to modelling users is to capture their expertise using implicit acquisition techniques and represent that expertise in terms of the conceptual model for the domain. These techniques are a set of methods that make use of information about users which can be extracted in an unobtrusive way from the working environment. Examples are the artifacts they have developed in the application domain, and the results of previous explanation dialogs between the user and the system [38].
Knowledge about the most common problems users have in using a system and about instructional strategies. This knowledge is required if someone wants to be a good coach or teacher and not only an expert. Explanations play a crucial role in instructional strategies. To generate good explanations is a more difficult problem in critic systems than in tutoring systems, because critic systems do not control the set of problems being addressed. Users learn best when they are situated in the actual context of their work and are able to receive explanations from an expert who can clear up misconceptions and clarify understanding. This helps users to restructure their own knowledge [44].

4. Prototypical systems

We have developed computer-based critics for several domains. By a careful analysis and detailed comparison of these system-building efforts, we developed general principles for designing critics and other intelligent support system. In this section, we describe one system, LISP-CRITIC, a system that critiques LISP code, in some detail and briefly describe two two other systems: FRAMER, a critic system for window based user interfaces, and JANUS, a system for architectural design. These three different systems were developed, because they require different domain knowledge: LISP-CRITIC "knows" about style in LISP programming, FRAMER "knows" about window-based interface design, and JANUS "knows about kitchen design.

4.1. LISP-CRITIC

LISP-CRITIC has evolved over the past three years from a knowledge-based program enhancement tool to a working context that we believe exemplifies the concept of a cooperative problem solving system. This evolution has resulted the development of four distinct systems (see fig. 3). Each version addressed an increasing set of research issues by building on what we learned from the previous version. The first three systems will be discussed in this section and the current version (which is under active development) in the next section.

CODE IMPROVER. The precursor to the LISP-CRITIC systems was CODE IMPROVER [2]. CODE IMPROVER functions as a knowledge-based post compiler, taking as input an executable FRANZLISP program and producing a version that either better facilitates human understanding by increasing cognitive efficiency or a version that executes more quickly by improving machine efficiency. The transformations the system used were captured in a knowledge base that was elicited from interviews with experienced LISP programmers. An example of the sort of rules contained in that knowledge-base is shown in fig. 4. CODE IMPROVER operated in a batch mode in the UNIX operating system environment.

CODE IMPROVER critiqued a user's code in the following ways:

- replace compound calls of LISP functions by simple calls to more powerful functions: *(not (evenp a))* may be replaced by *(oddp a)*;
- suggest the use of macros: *(setq a (cons b a))* may be replaced by *(push b a)*;
- find and eliminate 'dead' code: *(cond (...) (t...) (dead code))*;
- replace a copying (garbage generating) function

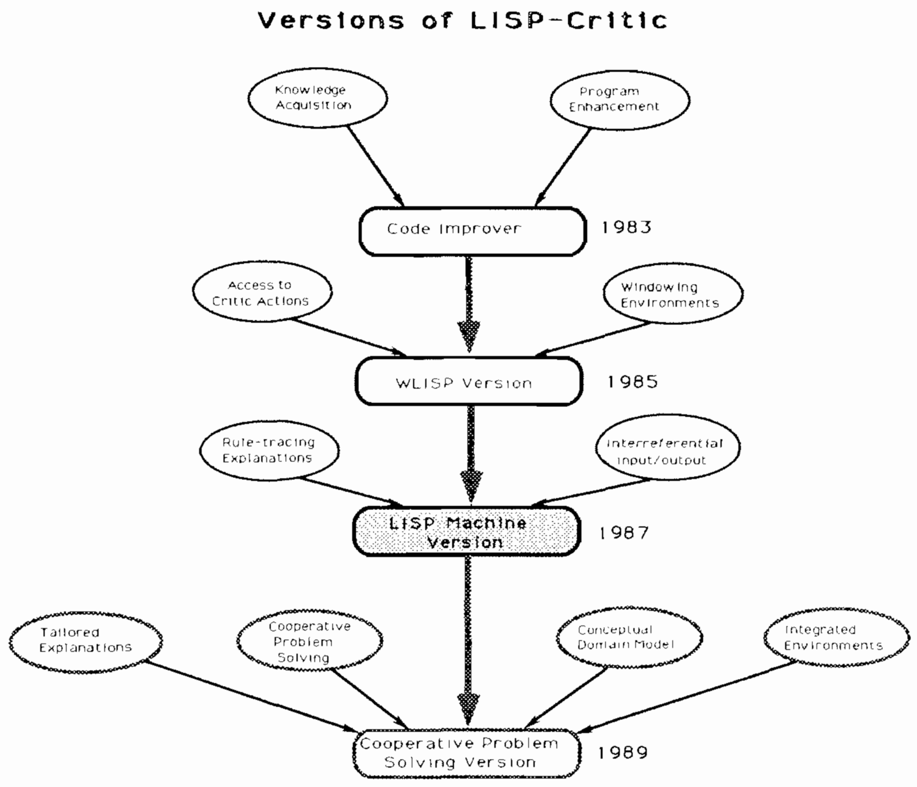


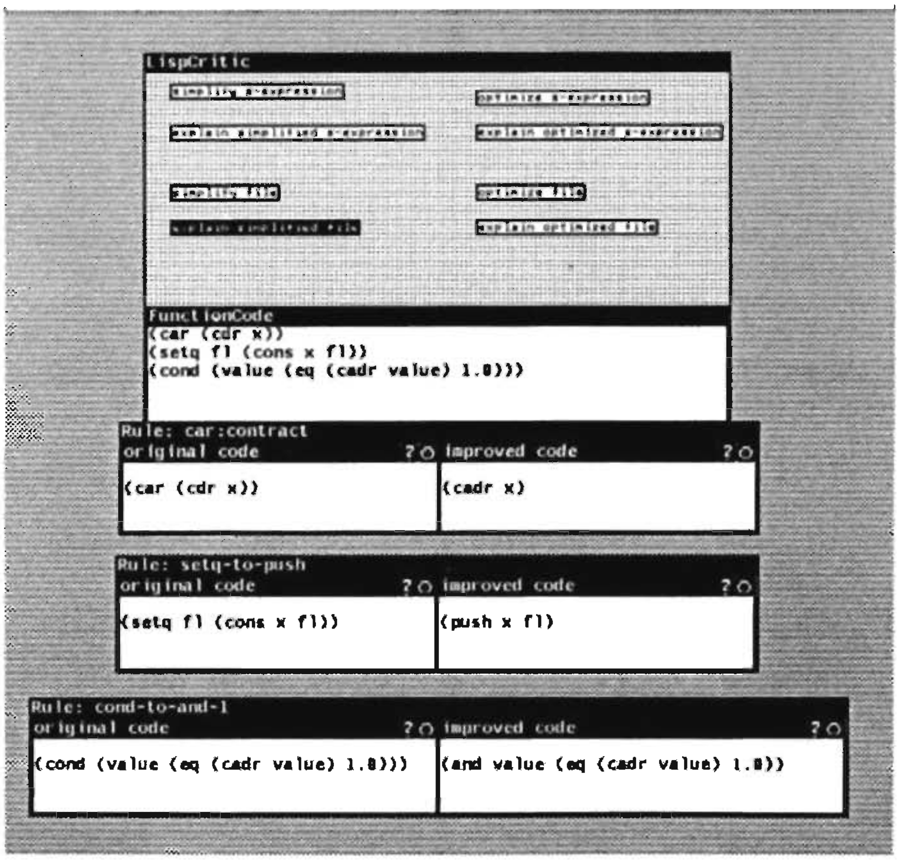
Fig. 3. The theoretical issues incorporated into versions of LISP-CRITIC. The versions of LISP-CRITIC are shown in the center of the above figure. Each of these was designed to address the specific theoretical issues that are indicated in the ovals. Current work is indicated by the objects that are grayed-out. The most recent fully operational version is the LISP Machine Version and the bottom theoretical issues are being incorporated into the Cooperative Problem Solving Version.

with a destructive function: (*append (explode word) chars*) may be replaced by (*nconc (explode word) chars*); see figs. 5 and 8;
- specialized functions: replace *equal* by *eq*;
- evaluate or partially evaluate expressions: (*sum a 3 b 4*) may be simplified to (*sum a b 7*).

WLISP VERSION. The first version of LISP-CRITIC [11] was designed to operate in the WLISP windowing environment on Bitgraph terminals, it is shown in fig. 4. It allows users to receive rudimentary explanations of the critic's suggestions in the form of rules that were invoked. Users can choose the kind of suggestions in which they are interested. This version was designed to take advantage of advances in human computer interaction techniques and to allow the user to learn from the system.

LISP MACHINE VERSION. In order to bring LISP-CRITIC closer to the working environment of real LISP programmers, it was integrated into a LISP Machine environment, the Symbolics 3600 Workstation. Fig. 6 shows the second version of LISP-CRITIC running as an activity in the Symbolics Genera Environment. The knowledge-base of LISP-CRITIC was updated to process COMMON LISP but the type of knowledge it contains and the way it applies that knowledge have remained relatively consistent.

The ideas incorporated into this version were aimed at trying to make the environment more



FunctionCode

```
(car (cdr x))
(setq fl (cons x fl))
(cond (value (eq (cadr value) 1.0)))
```

Rule: car:contract

original code ?	improved code ?
(car (cdr x))	(cadr x)

Rule: setq-to-push

original code ?	improved code ?
(setq fl (cons x fl))	(push x fl)

Rule: cond-to-and-1

original code ?	improved code ?
(cond (value (eq (cadr value) 1.0)))	(and value (eq (cadr value) 1.0))

Fig. 4. LISP-CRITIC in WLISP. This figure shows the LISP-CRITIC interface running in the WLISP windowing environment. The user can initiate an action by clicking a button. The FUNCTIONCODE pane displays the text of the program that the LISP-CRITIC is working on. The other three windows show transformations suggested by LISP-CRITIC. The "?" in the title line of the windows is the button for accessing the explanation system.

interactive for users. Some of these ideas were: providing users the ability to view and compare the two versions (their original code and the one generated by LISP-CRITIC) of the program in different windows, to request explanation of those differences, to access source code files in any local or remote directory available via network access, and to take advantage of many of the features for interreferential input/output provided by a LISP Machine environment. Explanations are provided at the rule-tracing level [5] and, if further clarification is required, pre-stored textual descriptions of rules in the knowledge-base are displayed.

Over the course of developing the three versions described above, the system has been used by two different user groups. One group consists of intermediate users who want to learn how to produce better LISP code. We have tested the usefulness of LISP-CRITIC for this purpose by gathering statistical data on the programs written by students in an introductory LISP course. The other group consists of experienced users who want to have their code "straightened out." Instead of refining their code by hand (which in principle these users can do), they use LISP-CRITIC to help them reconsider the code they have writ-

Replace a Copying Function with a Destructive Function

```
(rule append/.1-new.cons.cells-to-nconc/.1...     ;;; the name of the rule
   (?foo:{append append1}                         ;;; the original code
      (restrict ?expr                             ;;; condition
         (cons-cell-generating-expr expr))        ;;; (rule can only be applied
                                                  ;;; if "?expr" generates
                                                  ;;; cons cells)
      ?b)
   ==>
   ((compute-it:                                  ;;; the replacement
      (cdr (assq (get-binding foo)
                 '((append . nconc)
                   (append1 . nconc1)))))
    ?expr ?b)
   safe (machine))                                ;;; rule category
```

Example:

```
(append (explode word) chars)
==>
(nconc (explode word) chars)
```

Fig. 5. Example of a rule in the LISP-CRITIC. The rule "append/.1-new.cons.cell-to-nconc" replaces the function APPEND which generates a copy of the argument data structure in memory with the function NCONC which modifies the internal representation instead. This transformation is preferred in cases where users would like to minimize the use of memory and the freshly generated data structure is not used elsewhere in the program.

ten. The system has proven especially useful with code that is under development or frequently modified.

The architecture of LISP-CRITIC. The structure of the overall system is given in fig. 7. The user's code is analyzed and simplified according to the transformation rules. They contain information that is used to generate explanations. The user model obtains information from the rules that have fired, from the statistical analyzer, and from specialized knowledge acquisition rules which look for cues indicating that a specific concept of LISP is either known or not known by the user. In return, the user model determines which rules should fire and what explanations should be generated.

Support for understanding the critique. The use of LISP-CRITIC by students has shown that the critique given is often not understood. Therefore we use additional system components to illustrate and explain the LISP-CRITIC'S advice. KAESTLE, a visualization tool that is part of our software oscilloscope [3], allows us to illustrate the functioning and validity of certain rules. In fig. 8, we use KAESTLE to show why the transformation

$$(append\ (explode\ word)\ chars) \Rightarrow (nconc\ (explode\ word)\ chars)$$

is safe (because explode is a cons-generating function; see fig. 4), whereas the transformation

$$(append\ chars\ (explode\ word)) \Rightarrow (nconc\ chars\ (explode\ word))$$

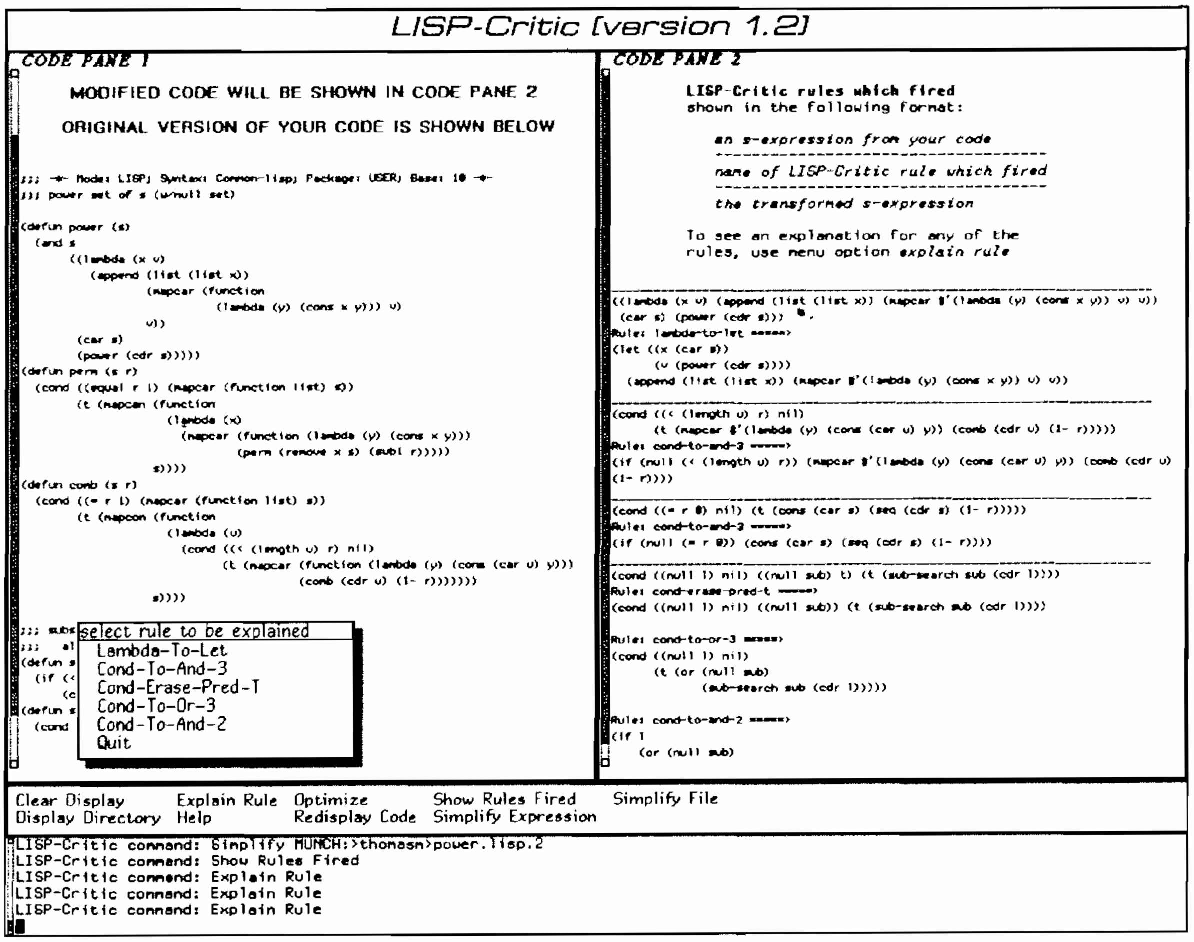
is unsafe because the destructive change of the value of the first argument by nconc may cause undesirable side effects.

LISP-CRITIC has proven to be a useful tool as well as an interesting environment in which to address issues that are at the heart of our entire research effort. LISP-CRITIC allows us to explore cooperative problem solving in an environment which is based on an available formalized representation of the problem domain, the LISP programming language.

4.2. FRAMER

FRAMER [34;16] is a design environment for the construction of window-based user interfaces. Design environments reduce the amount of knowledge designers have to acquire before they can do useful work. The FRAMER system (see fig. 9) is an enhanced version of the Symbolics FRAME-UP tool. These systems permit users to design their own user interfaces without writing code.

Representation of design knowledge as critics. FRAMER contains design knowledge represented as



LISP-Critic [version 1.2]

CODE PANE 1

MODIFIED CODE WILL BE SHOWN IN CODE PANE 2

ORIGINAL VERSION OF YOUR CODE IS SHOWN BELOW

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;; power set of s (w/null set)

(defun power (s)
  (and s
       ((lambda (x v)
          (append (list (list x))
                  (mapcar (function
                            (lambda (y) (cons x y))) v)
                  v))
        (car s)
        (power (cdr s)))))
(defun perm (s r)
  (cond ((equal r 1) (mapcar (function list) s))
        (t (mapcan (function
                     (lambda (x)
                       (mapcar (function (lambda (y) (cons x y)))
                               (perm (remove x s) (sub1 r)))))
                  s))))
(defun comb (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (mapcar (function (lambde (y) (cons (car u) y)))
                                        (comb (cdr u) (1- r)))))))
                  s))))

;;; subs
;;;   al
(defun s
  (if (<
     (c
(defun s
  (cond
```

select rule to be explained
Lambda-To-Let
Cond-To-And-3
Cond-Erase-Pred-T
Cond-To-Or-3
Cond-To-And-2
Quit

CODE PANE 2

LISP-Critic rules which fired
shown in the following format:

an s-expression from your code
name of LISP-Critic rule which fired
the transformed s-expression

To see an explanation for any of the rules, use menu option *explain rule*

```
((lambda (x v) (append (list (list x)) (mapcar #'(lambda (y) (cons x y)) v) v))
 (car s) (power (cdr s)))
Rule: lambda-to-let =====>
(let ((x (car s))
      (v (power (cdr s))))
  (append (list (list x)) (mapcar #'(lambda (y) (cons x y)) v) v))
--------------------------------------------------------------------
(cond ((< (length u) r) nil)
      (t (mapcar #'(lambda (y) (cons (car u) y)) (comb (cdr u) (1- r)))))
Rule: cond-to-and-3 =====>
(if (null (< (length u) r)) (mapcar #'(lambda (y) (cons (car u) y)) (comb (cdr u)
(1- r))))
--------------------------------------------------------------------
(cond ((= r 0) nil) (t (cons (car s) (seq (cdr s) (1- r)))))
Rule: cond-to-and-3 =====>
(if (null (= r 0)) (cons (car s) (seq (cdr s) (1- r))))
--------------------------------------------------------------------
(cond ((null l) nil) ((null sub) t) (t (sub-search sub (cdr l))))
Rule: cond-erase-pred-t =====>
(cond ((null l) nil) ((null sub)) (t (sub-search sub (cdr l))))

Rule: cond-to-or-3 =====>
(cond ((null l) nil)
      (t (or (null sub)
             (sub-search sub (cdr l)))))

Rule: cond-to-and-2 =====>
(if l
    (or (null sub)
```

```
LISP-Critic command: Simplify MUNCH:>thomasm>power.lisp.2
LISP-Critic command: Show Rules Fired
LISP-Critic command: Explain Rule
LISP-Critic command: Explain Rule
LISP-Critic command: Explain Rule
```

Fig. 6. The LISP-CRITIC interface on the symbolics computer. This figure shows the LISP-CRITIC version 2 interface on the Symbolics 3600. The user can request LISP-CRITIC critique a program code file or can enter any LISP expression and receive suggestions as to How to improve it. To facilitate user understanding of LISP-CRITIC's suggestions, explanations in the form of rule tracing are available as are more specific explanations of those rules. In the figure shown above the user has submitted a program file for critiquing, is being shown the trace of the rules that fired and is about to request an explanation for the rationale behind one of these rules from a pop-up menu.

a set of critics. The critics can be invoked by selecting the *Suggest Improvements* menu item or by typing the *Suggest Improvements* command. In the example of fig. 9, three critics fire and display suggestions in text surrounded by a rectangle in the dialog pane. Suggestions are active, mouse-sensitive objects, and two operations, *Explain* and *Remedy*, are available on them (see mouse documentation line at the bottom of the screen). One of the critics suggested moving the title pane, which was put at the bottom of the interface, to the top of the window frame. By clicking the left mouse button on this suggestion, the designer can obtain an explanation for this suggestion.

For some suggestions, a *Remedy* operation is available; that is, FRAMER itself can fix a problem it has detected. For example, clicking the middle mouse button on the suggestion about the title pane would cause the system to move the pane to the top of the frame. The *Remedy* operation can be considered an expert system that implements improvements that can be executed without additional user intervention.

The system not only gives negative criticism but can also describe positive features of the design with the *Praise* command (see last command in fig. 9). This positive feedback reassures designers that they are on the right track and helps

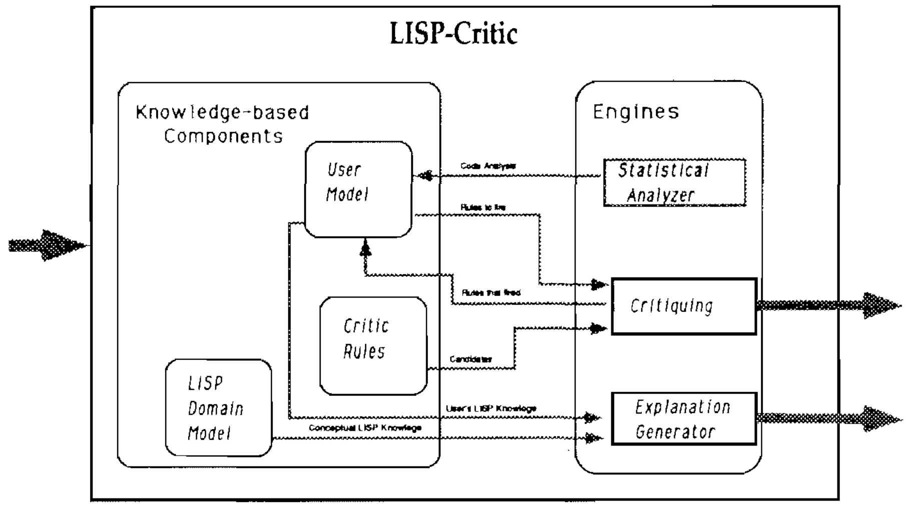
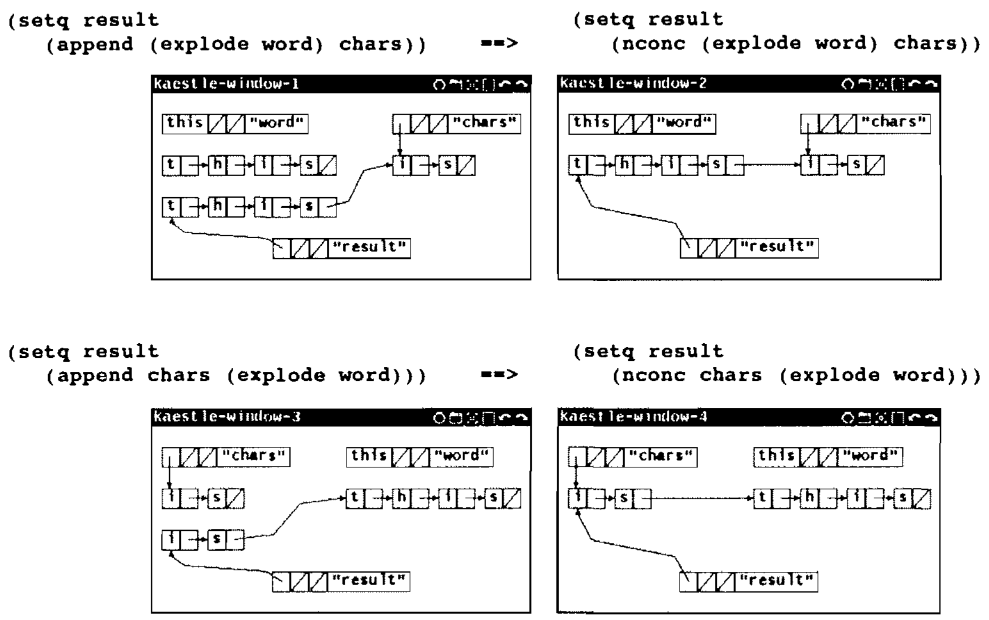


Fig. 7. The architecture of LISP-CRITIC. This figure shows the internal components of LISP-CRITIC and the information flow between them.

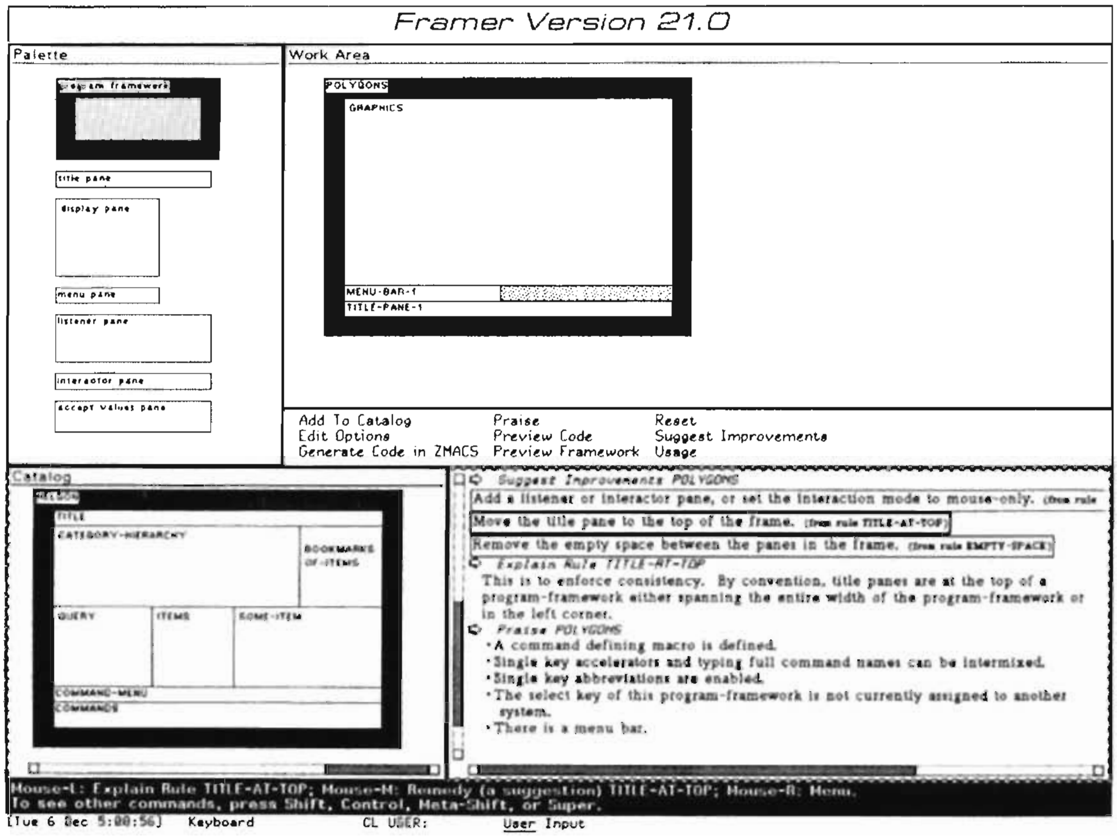
them preserve the good characteristics in ongoing modifications.

The critic knowledge base contains rules about naming the program, arranging window panes, specific knowledge about title panes, dialog panes, and menu panes, and knowledge about invoking a


```
(setq result                                  (setq result
   (append (explode word) chars))   ==>          (nconc (explode word) chars))
```

```
(setq result                                  (setq result
   (append chars (explode word)))   ==>          (nconc chars (explode word)))
```

Fig. 8. Illustration of the validity of a rule using Kaestle. The KAESTLE system is showing the user the effects of the applying the APPEND and the NCONC functions to the same data structure. The motivation behind this display is to demonstrate that although these two functions return the same result they have different internal effects on the stored data structures. In the example shown, the variable word is bound to the value this and the variable chars is bound to the list (i s).



Suggest Improvements POLYGONS
Add a listener or interactor pane, or set the interaction mode to mouse-only. (from rule
Move the title pane to the top of the frame. (from rule TITLE-AT-TOP)
Remove the empty space between the panes in the frame. (from rule EMPTY-SPACE)
Explain Rule TITLE-AT-TOP
This is to enforce consistency. By convention, title panes are at the top of a program-framework either spanning the entire width of the program-framework or in the left corner.
Praise POLYGONS
- A command defining macro is defined.
- Single key accelerators and typing full command names can be intermixed.
- Single key abbreviations are enabled.
- The select key of this program-framework is not currently assigned to another system.
- There is a menu bar.

Mouse-L: Explain Rule TITLE-AT-TOP; Mouse-M: Remedy (a suggestion) TITLE-AT-TOP; Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 6 Dec 5:00:56] Keyboard CL USER: User Input

Fig. 9. FRAMER. In the figure, the following components can be seen (clockwise from the top left): a palette of parts, a work area, a menu, a dialog pane, and a catalog. An interface is built in a direct-manipulation interaction style. Parts are obtained from the palette and combined in the work area. An interface can be built from the parts in the palette or, alternately, it can be created by selecting and modifying an existing interface in the catalog. The catalog is a scrollable window that contains typical designs.

program and selecting interaction modes. This set of rules is based on a study of existing systems in our computing environment. Some of the rules represent system constraints, for example, that a window frame must be completely divided up into panes. Other rules are concerned with the consistency between different applications and functional grouping.

Fig. 10 shows a typical critic rule. This rule contains knowledge about the relationship of the interaction mode and the configuration of window panes in the interface. If the *mouse-and-keyboard* interaction mode is selected, then the rule suggests adding a dialog pane. A *Remedy* action is also defined. Invoking the *Remedy* operation associated with this rule causes the system to add a listener pane at the bottom of the window frame.

Critics also operate on the designs stored in the *CATALOG*. These designs can be praised and critiqued, and when brought into the work area, they can be modified and used as a starting point for redesign. This feature is important for educational settings where students can study the critique the system generates for learning examples.

4.3. JANUS: A cooperative system for kitchen design

JANUS [22;21] allows designers to construct artifacts in the domain of architectural design and at the same time informs them about principles of design and their underlying rationale by integrating two design activities: construction and argumentation. *Construction* is supported by a knowledge-based graphical design environment (see Fig. 11) and *argumentation* is supported by a hypertext system (see fig. 12).

```
;; A critic rule named need-dialog-pane. The rule applies to program frameworks.

(define-critic-rule need-dialog-pane program-framework
  ;; Applicability condition. This rule is applicable if the
  ;; interaction mode is mouse-and-keyboard.
  :applicability (eq (interaction-mode self) :mouse-and-keyboard)

  ;; The rule is violated if there is no pane of type dialog-pane in
  ;; the set on inferiors of a program framework.
  :condition (null (find-if (pane-of-type 'dialog-pane) inferiors))

  ;; The Remedy operation adds a listener-pane.
  :remedy (let ((pane (make-instance 'listener-pane :x (+ x 20) :y (+ y 184)
                                                     :superior self)))
            (add-inferior self pane)
            (display-icon pane))

  ;; Text of the suggestion made to the user if critic is applicable.
  :suggestion "Add a listener or interactor pane, or set the interaction
  mode to mouse-only."

  ;; Text for Praise command.
  :praise "There is a listener or interactor pane."

  ;; Text for Explain command.
  :explanation "Since the interaction mode is mouse-and-keyboard, a dialog pane is
               required for typing in commands.")
```

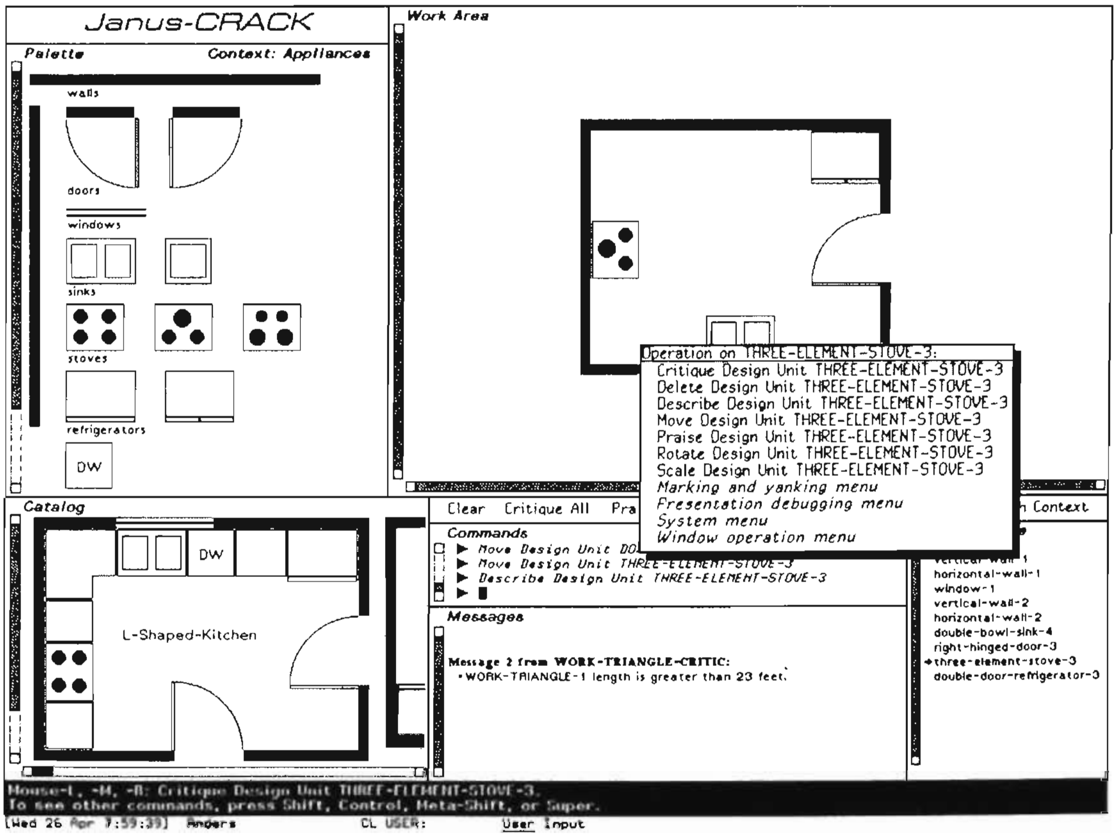
Fig. 10. An example of a critic rule. The rule "need-dialog-pane" applies to program frameworks. The rule suggests adding a listener or interactor pane if an interaction mode mouse-and-keyboard was specified.

In a fashion similar to FRAMER, JANUS provides a set of domain-specific building blocks and has knowledge about how to combine them into useful designs. With this knowledge it "looks over the shoulder" of users carrying out a specific design. If it discovers a shortcoming in the users' designs, it provides a critique, suggestions, and explanations, and assists users in improving their designs. JANUS is not an expert system that dominates the process by generating new designs from high-level goals or resolving design conflicts automatically. Users control the behavior of the system at all times (e.g., the critiquing can be "turned on and off"), and if users disagree with JANUS, they can modify its knowledge base.

Critics in JANUS are procedures for detecting non-satisficing partial designs. JANUS' concept for integrating the constructive and argumentative component originated from the observation that the critiques are a limited type of argumentation. The construction actions can be seen as attempts to resolve design issues. For example, when a designer is positioning the sink in the kitchen, the issue being resolved is "Where should the sink be located"?

The knowledge-based critiquing mechanism in JANUS bridges the gap between construction and argumentation. This means that critiquing and argumentation can be coupled by using JANUS' critics to provide the designer with immediate entry into the place in the hypertext network containing the argumentation relevant to the current construction task. Such a combined system provides argumentative information for construction effectively, efficiently, and designers do not have to realize before hand that information will be required, anticipate what information is in the system or know how to retrieve it.

JANUS' construction component. The constructive part of JANUS supports the building of an artifact either "from scratch" or by modifying an existing design. To construct from scratch, the designer chooses building blocks from a design units "Palette" and positions them in the "Work area" (see fig. 11).


Janus-CRACK
Palette
Context: Appliances
walls
doors
windows
sinks
stoves
refrigerators
DW
Work Area
Operation on THREE-ELEMENT-STOVE-3:
Critique Design Unit THREE-ELEMENT-STOVE-3
Delete Design Unit THREE-ELEMENT-STOVE-3
Describe Design Unit THREE-ELEMENT-STOVE-3
Move Design Unit THREE-ELEMENT-STOVE-3
Praise Design Unit THREE-ELEMENT-STOVE-3
Rotate Design Unit THREE-ELEMENT-STOVE-3
Scale Design Unit THREE-ELEMENT-STOVE-3
Marking and yanking menu
Presentation debugging menu
System menu
Window operation menu
Catalog
L-Shaped-Kitchen
Clear
Critique All
Pra
Context
Commands
Messages
horizontal-wall-1
window-1
vertical-wall-2
horizontal-wall-2
double-bowl-sink-4
right-hinged-door-3
three-element-stove-3
double-door-refrigerator-3

```
▶ Move Design Unit DO
▶ Move Design Unit THREE-ELEMENT-STOVE-3
▶ Describe Design Unit THREE-ELEMENT-STOVE-3
▶ ▮
```

```
Message 2 from WORK-TRIANGLE-CRITIC:
 •WORK-TRIANGLE-1 length is greater than 23 feet.
```

```
Mouse-L, -M, -R: Critique Design Unit THREE-ELEMENT-STOVE-3.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Wed 26 Apr 7:59:39]  Anders                CL USER:        User Input
```

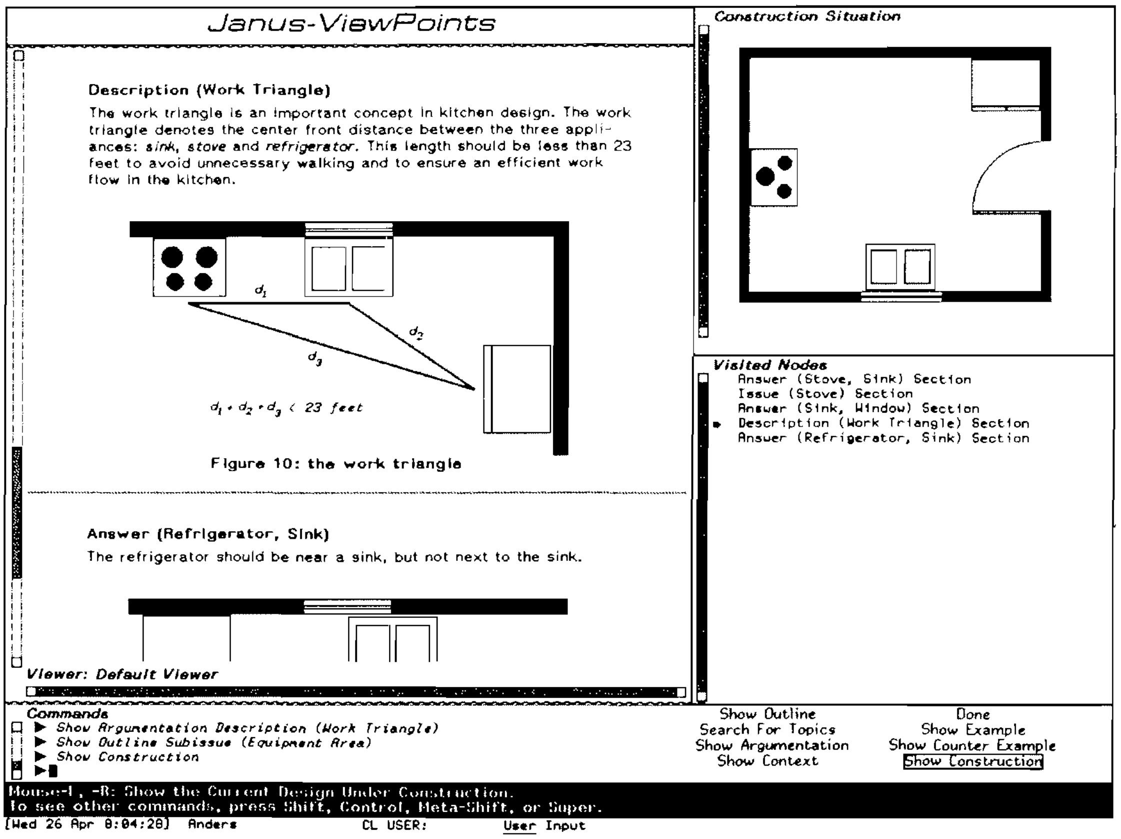
Fig. 11. JANUS construction interface The interface of JANUS'S construction component is based on the world model. Design units are selected from the Palette, and moved into the work area. Operations on design units are available through menus. The screen image shown displays a message from the WORK-TRIANGLE-CRITIC.

To construct by modifying an existing design, the designer uses the "CATALOG" (lower left in fig. 11), which contains several example designs. The designer can browse through this catalog of examples until an interesting one is found. This design can then be selected and brought into the "Work Area", where it can be modified.

The CATALOG contains both "good" designs and "poor" designs. The former satisfy all the rules of kitchen design and will not generate a critique. People who want to design without having to bother with knowing the underlying principles might want to select one of these, since minor modifications of them will be probably result in few or no suggestions from the critics. The "poor" designs in the CATALOG support learning the design principles. By bringing these into the "Work Area", users can subject them to critiquing and thereby illustrate those principles of kitchen design that are known to the system.

The "good" designs in the CATALOG can also be used to learn design principles and explore their argumentative background. This can be done by bringing them into the "Work Area" then using the "Praise all" command. This command causes the system to display all of the rules that the selected example satisfies. This positive feedback also provides entry points into the hypertext argumentation.

JANUS' argumentation component. The hypertext component of Janus is implemented using Symbolics Concordia and Document Examiner software. Concordia is a hypertext editor [62] with which the issue base is implemented. The Document Examiner [61] provides functionality for on-


Janus-ViewPoints

Description (Work Triangle)

The work triangle is an important concept in kitchen design. The work triangle denotes the center front distance between the three appliances: *sink*, *stove* and *refrigerator*. This length should be less than 23 feet to avoid unnecessary walking and to ensure an efficient work flow in the kitchen.

Figure 10: the work triangle

Answer (Refrigerator, Sink)

The refrigerator should be near a sink, but not next to the sink.

Viewer: Default Viewer

Construction Situation

Visited Nodes
```
Answer (Stove, Sink) Section
Issue (Stove) Section
Answer (Sink, Window) Section
Description (Work Triangle) Section
Answer (Refrigerator, Sink) Section
```

Commands
```
▶ Show Argumentation Description (Work Triangle)
▶ Show Outline Subissue (Equipment Area)
▶ Show Construction
▶
```

```
Mouse-L, -R: Show the Current Design Under Construction.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Wed 26 Apr 8:04:28] Anders          CL USER:        User Input
```

Fig. 12. JANUS argumentation interface. JANUS's argumentation component uses the Symbolics Document Examiner as a delivery interface. The construction situation can be displayed in one of the panes to allow users to inspect the constructive and argumentative context simultaneously.

line presentation and browsing of the issue base by users.

When users enter the argumentative part of JANUS, they are brought into a section of the issue base relevant to their current construction situation. Their point of entry into the hypertext network should contain the information required to understand the issue of interest. But argumentation on an issue can be large and complex so they can use this initial display of relevant information as a starting place for a navigational journey through the issue base, following links that will lead them to additional information. Upon completion of the examination of the argumentative information the designer can return to construction and complete the current task.

Critics as Hypertext Activation Agents. JANUS' knowledge-based critics serve as the mechanism to link construction with argumentation. They "Watch over the shoulders" of designers, displaying their critique in the "Messages" pane (center bottom in fig. 11) when design principles are violated. In doing so they also identify the argumentative context which is appropriate to the current construction situation.

For example, when a designer has designed the kitchen shown in fig. 11, the "Work-Triangle-Critic" fires and detects that the work triangle is too large. To see the arguments surrounding this issue, the designer has only to click on the text of this criticism with the mouse. The argumentative context shown in fig 12 is then displayed.

5. Extending the critiquing paradigm to support cooperative problem solving

The long-term goal of this effort is to develop the full potential of the critic paradigm and to make it a prototype for designing cooperative problem solving systems. In this section, we will first discuss the evaluations of the systems described in the previous section. The results of these evaluations have led to the articulation of additional requirements for critic systems and design environments which bring them one step closer acting as cooperative problem solving systems.

5.1. Evaluation

We have tested our critics systems with real users over extended periods of time. Various evaluation methods (e.g., thinking-aloud protocols [35] and questionnaires) showed that a strictly quantitative evaluation is not feasible because many important factors are only qualitative.

Results of evaluation of LISP-CRITIC. The results of our evaluations of LISP-CRITIC showed its strengths and weaknesses.

Some of the *strengths* of LISP-CRITIC are:

- It supports users in doing their own tasks and it supports intermediate users, not just beginners;
- It enhances incremental learning;
- It fosters reusability by pointing out operations that exist in the system;
- It can be applied to every program (in the worst case the system does not find anything to complain about);
- It is not just a toy system because users have used it in the context of their everyday work;
- Using it does not require users to provide information in addition to the code.

Some of the *weaknesses* of LISP-CRITIC are:

- It use only low-level transformations (i.e., it operates primarily at the level of s-expressions);
- It has no understanding of the problem the user is trying to solve; this limits analysis because LISP-CRITIC cannot distinguish between constructs the user does not know and those not required to solve the problem.
- The rules are not tied to higher-level conceptual units;
- The explanations should be generated more dynamically [43].

Evaluation of FRAMER and JANUS. The evaluation of FRAMER (which provided the design rationale for several different versions of the system) is described in detail in [34].

In our evaluation of JANUS we accumulated feedback about its strengths and shortcomings. One of our colleagues who is not a professional kitchen designer, had just remodeled his kitchen. He considered JANUS a valuable tool. The critiques generated by the system during his design process illustrated several design concepts of which he was not aware. In addition to generating a specific design for his kitchen, our colleague increased his general knowledge about kitchen design.

The system was also used by a design methodologist who considered the cooperative, user-dominated approach of JANUS its most important feature. He felt that this feature set JANUS apart from expert system oriented design tools that users have little control over and that often reduce users to spectators of the system's operations. We have deliberately avoided equipping the current version of JANUS with independent design capabilities. Too much assistance and too many automatic procedures can reduce the users' motivation by not providing sufficient challenge.

In contrast to most current CAD systems, that are merely drafting tools rather than design tools, JANUS has some "understanding" of the design space. This knowledge allows the system to critique a design during the design process – a capability absent in CAD systems.

5.2. Additional requirements

As a result of building multiple versions of LISP-CRITIC, the FRAMER and JANUS systems we have developed a general schema for the architecture of a knowledge-based critic. Here we describe the role of and general requirements for each component of that architecture: the domain knowledge base, the user model, and the explanation component.

Domain knowledge. Initial versions of our critics incorporated a rule-based representation of domain knowledge. These rules analyzed the user's product (or partial product). This representation is both efficient to implement and apply. However, when we sought to improve our systems in order to provide explanations, fundamental limitations were recognized. In order to extend the cooperative behavior of the system, a representation of domain knowledge based upon a conceptual structure of that particular domain is needed. In the case of LISP-CRITIC that representation follows the notation for conceptual graphs [53] and is implemented using an object oriented paradigm, the COMMON LISP OBJECTS SYSTEM. This conceptual structure is used by the explanation component to determine which fundamental concepts have to be described to explain a new concept to the user or to find related concepts to use in a differential description approach.

Model of the user. In order to extend our computer-based critics toward cooperative problem solving, they must contain a user model. It is possible to develop a usable system without an underlying user modelling component but instead use a default user model that is "designed-into" the systems. The default model causes the system to treat all users the same. The specific questions that have to be solved in order to achieve a usable user model are how to represent the knowledge of each user and how to acquire and update that knowledge over time.

Representation of the user model. Our first attempts to model the user were classification approaches based on observations of the users' programming habits. This approach turned out to be inadequate. In order to model domain expertise, knowledge needs to be represented in the user model as a collection of concepts that each individual knows. A whole class of users will not know the same set of concepts just because they have the same background or experience. A survey of experienced LISP programmers in our department confirmed this intuition. Our test of expertise was the programmers' understanding of generalized variables in COMMON LISP [55] and their preference for using and teaching the "setq" and "setf" special forms. We discovered a significant variability not only in these experts' preferences but also in their understanding of the concept. These insights have led us to represent users as a collection of concepts that they know or do not know about LISP along with an associated confidence factor.

Acquisition of the user model. We attempt to acquire the user model with implicit acquisition techniques. These are represented as a collection of methods that operate on the user model making use of information available in the environment (e.g., the programmer's code) and the domain model. The entire system is based on an object-oriented approach allowing these methods to be defined for classes of domain objects and the user model. Some of our methods are based on the results of analysis of the code, some track the explanations the user has requested and received in the past, some infer the users' preference by tracking the suggestions they have rejected and others periodically check the model for internal consistency.

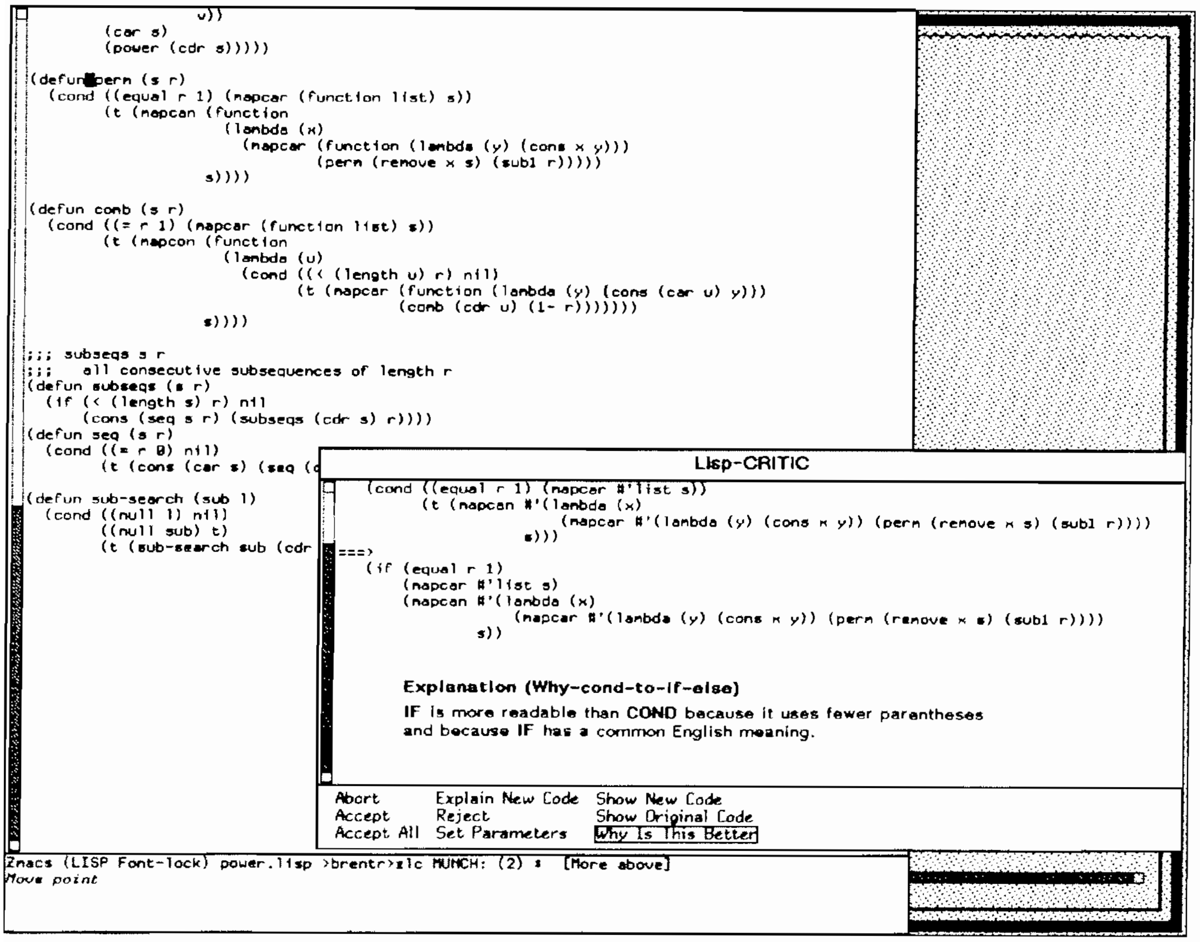
Some of the methods incorporated into our acquisition methodology are the result of an empirical study of how expert human programmers accomplish the same task. Experts were provided samples of code from student programmers and asked to make inferences about the expertise level of each student. The protocols were analyzed with an eye toward what cues human experts look for in the code and what inference they make when they find a particular cue [20]. One approach that we attempted to incorporate was the use of stereotyping [50]. We experienced fundamental limitations with this approach because no prescribed stereotypes of LISP programmers exist, nor are there guaranteed techniques for acquiring stereotypes from statistical analysis of a sample population. The primary source for cues is LISP-CRITIC rules that fire when a pattern is found in the user's code. Collections of rules that have fired imply that the programmer knows a particular concept.

Explanations. The focus of our explanations is to overcome any misunderstanding the user might have about the critic's suggestions. Our research efforts have determined that these explanations should meet three criteria:

- Explanations should be tailored to the expertise of the individual.
- A good metric keeps explanations as concise as possible.
- Explanation strategies should plan for the all to likely case where the initial attempt fails to satisfy a user. Backup techniques are needed.

We achieve the first criterion using the concept-based domain model to determine what concepts need to be explained to understand a particular suggestion and then consulting the user model to determine which of those concepts the user requesting the explanation already knows. This helps to achieve a concise explanation. The conciseness metric results from applying the theory of discourse comprehension to our explanation task [32]. In the case where the system provides an inadequate explanation, the need for a backup technique is fulfilled by presenting explanations in a hypertext format that provides the user access to a rich on-line database of documentation about the application domain. We have termed our overall strategy "minimalist explanations" [19].

The capability of the user to access the rationale behind the suggestions of a critic has developed into one of our central issues. We have or are in the process of incorporating several techniques to improve explanations, these include hypertext, the argumentation methodology, and the 'minimalist approach'. Whether one or some combination of these stands out as an ideal solution remains an open question, one that can only be answered through an empirical evaluation of the implemented systems. In addition we have ideas



```
                    v))
         (car s)
         (power (cdr s)))))

(defun perm (s r)
  (cond ((equal r 1) (mapcar (function list) s))
        (t (mapcan (function
                     (lambda (x)
                       (mapcar (function (lambda (y) (cons x y)))
                               (perm (remove x s) (sub1 r)))))
                   s))))

(defun comb (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (mapcar (function (lambda (y) (cons (car u) y)))
                                        (comb (cdr u) (1- r)))))))
                   s))))

;;; subseqs s r
;;;   all consecutive subsequences of length r
(defun subseqs (s r)
  (if (< (length s) r) nil
      (cons (seq s r) (subseqs (cdr s) r))))
(defun seq (s r)
  (cond ((= r 0) nil)
        (t (cons (car s) (seq (c

(defun sub-search (sub l)
  (cond ((null l) nil)
        ((null sub) t)
        (t (sub-search sub (cdr
```

Lisp-CRITIC

```
  (cond ((equal r 1) (mapcar #'list s))
        (t (mapcan #'(lambda (x)
                         (mapcar #'(lambda (y) (cons x y)) (perm (remove x s) (sub1 r))))
                    s)))
===>
  (if (equal r 1)
      (mapcar #'list s)
      (mapcan #'(lambda (x)
                     (mapcar #'(lambda (y) (cons x y)) (perm (remove x s) (sub1 r))))
               s))
```

Explanation (Why-cond-to-if-else)

IF is more readable than COND because it uses fewer parantheses and because IF has a common English meaning.

```
Zmacs (LISP Font-lock) power.lisp >brentr>zlc MUNCH: (2) *   [More above]
Move point
```

Fig. 13. The LISP-CRITIC in ZMACS on the Symbolics computer. This figure shows LISP-CRITIC running within the ZMACS editor on a Symbolics 3600. The user invokes LISP-CRITIC by positioning the cursor within the context of a function definition that he or she would like to have critiqued. LISP-CRITIC displays its suggestions to the user one transformation at a time. The user can accept, reject or request an explanation for each transformation. In the screen image above the user has asked LISP-CRITIC to explain why the COND-TO-IF transformation is suggested. The explanation is displayed in a hypertext form, the user can select IF or COND with the mouse and access additional information stored in the Document Examiner's on-line documentation.

for still other approaches to enhance explanatory capabilities that have not yet been fully developed.

- *Differential descriptions* is an approach that depends heavily on the user model and on maintaining a record of the user's work. Descriptions of concepts new to a particular user will be generated by relating them to concepts already known to a user; these are contained in the user model.
- *Graphical techniques* can help the user to see how his work relates to the entire product and likewise how the critic's suggestions are integrated. Other systems use graphical techniques to conceptualize the program development process and integrate this with explanatory capabilities [47;46]. This type of additional capability could integrate nicely into the environment in which our critics now operate.

5.3. A new version of the LISP-CRITIC

The components discussed in the previous section and the results of our evaluations are being used in the most recent implementation of LISP-CRITIC. However, simply adding functionality to LISP-CRITIC is not a sufficient condition to achieve our goal of developing a cooperative problem solving system, the manner in which that functionality is achieved also plays a major role. We will describe the current version of LISP-CRITIC and some of the ideas that will be tested in that implementation.

The most recent version of LISP-CRITIC also runs on Symbolics LISP Machines, however it has been integrated into the programming development environment, ZMACS. Having to use LISP-CRITIC as an activity separate from the programmers working context was considered one of the major shortcomings of the previous versions. Fig. 13 shows a screen image of the system invoked from the ZMACS editing environment. The general idea is that programmers working on LISP program code reaches a point where they would appreciate an "outsider" looking it over and giving suggestions as to how it might be improved. The granularity of the critique was reduced from the file level to the scope of a function definition.

Users decide whether to accept or reject the transformation for each particular suggestion of the critic. If they choose to accept the transformation then the system automatically replaces that piece of code in the users' buffer, similar to the *Remedy* option in FRAMER. Previously users had to either accept the version of the program created by LISP-CRITIC or individually make changes to the original version of the program. This version of LISP-CRITIC has been designed to include representations of domain knowledge for use in critiquing and in explanation-giving/user modelling, a model of individual users, and an explanation component.

6. Conclusions

Computer-based critics incorporate powerful ideas from human computer communication and artificial intelligence research into a system that combines the best aspects of human and computational cognition. Critics have the potential to foster a cooperative relationship in support of learning and working between a user and a knowledge-based system.

Implementation of this concept requires that critics contain domain knowledge represented in a form that is applicable both to problem solving and to explanations. An explanation component uses the knowledge base and information contained in the user model to generate explanations. The system must be able to share its domain knowledge with the user, and it must construct an individual user model to support this sharing process.

We have developed several critic systems that incorporate these ideas. The successes and failures of these efforts is helping us to define the characteristics and design considerations for critic systems as well as to gauge their potential. These results are applicable to the entire class of cooperative problem solving systems.

References

[1] J.R. Anderson, B.J. Reiser. The LISP Tutor. BYTE 10(4): 159–175, April, 1985.

[2] H.-D. Boecker. Softwareerstellung als wissensbasierter Kommunikations- und Designprozess. Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, April, 1984.

[3] H.-D. Boecker, G. Fischer, H. Nieper. The Enhancement of Understanding Through Visual Representations. In

Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), pages 44–50. ACM, New York, April, 1986.

[4] B.G. Buchanan, E.H. Shortliffe. Human Engineering of Medical Expert Systems. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley Publishing Company, Reading, MA, 1984, pages 599–612, Chapter 32.

[5] B.G. Buchanan, E.H. Shortliffe. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley Publishing Company, Reading, MA, 1984.

[6] R.R. Burton, J.S. Brown. An Investigation of Computer Coaching for Informal Learning Activities. In D.H. Sleeman, J.S. Brown (editors), Intelligent Tutoring Systems, chapter 4, pages 79–98. Academic Press, London–New York, 1982.

[7] J.G. Carbonell. User Modeling and Natural Language Interface Design. In H. Balzert (editor), Software-Ergonomie. Tagung I/1983 des German Chapter of the ACM, April 1983, Nuernberg, pages 21–29. Teubner Verlag, Stuttgart, 1983.

[8] J.M. Carroll, J. McKendree. Interface Design Issues for Advice-Giving Expert Systems. Communications of the ACM 30(1): 14–31, January, 1987.

[9] D.F. Dansereau. Cooperative Learning Strategies. Learning and Study Strategies: Issues in Assessment, Instruction and Evaluation. Academic Press, New York, 1988, pages 103–120, Chapter 7.

[10] S.W. Draper. The Nature of Expertise in UNIX. In Proceedings of INTERACT'84, IFIP Conference on Human-Computer Interaction, pages 182–186. Elsevier Science Publishers, Amsterdam, September, 1984.

[11] G. Fischer. A Critic for LISP. In J. McDermott (editor), Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), pages 177–184. Morgan Kaufmann Publishers, Los Altos, CA, August, 1987.

[12] G. Fischer. Cognitive View of Reuse and Redesign. IEEE Software, Special Issue on Reusability 4(4): 60–72, July, 1987.

[13] G. Fischer. Cooperative Problem Solving Systems. In Proceedings of the 1st Simposium Internacional de Inteligencia Artificial (Monterrey, Mexico), pages 127–132. October, 1988.

[14] G. Fischer, W. Kintsch, P.W. Foltz, S.M. Mannes, H. Nieper-Lemke, C. Stevens. Theories, Methods, and Tools for the Design of User-Centered Systems (Interim Project Report, September 1986-February 1989). Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, March, 1989.

[15] G. Fischer, A.C. Lemke. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. Human-Computer Interaction 3(3): 179–222, 1988.

[16] G. Fischer, A.C. Lemke. Knowledge-Based Design Environments for User Interface Design. Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, March, 1989.

[17] G. Fischer, A.C. Lemke, T. Schwab. Knowledge-Based Help Systems. In Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), pages 161–167. ACM, New York, April, 1985.

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

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

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

[21] G. Fischer, R. McCall, A. Morch. Design Environments for Constructive and Argumentative Design. In Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), pages 269–275. ACM, New York, May, 1989.

[22] G. Fischer, A. Morch. CRACK: A Critiquing Approach to Cooperative Kitchen Design. In Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), pages 176–185. ACM, New York, June, 1988.

[23] G. Fischer, H. Nieper-Lemke. HELGON: Extending the Retrieval by Reformulation Paradigm. In Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), pages 357–362. ACM, New York, May, 1989.

[24] G. Fischer, M. Schneider. Knowledge-Based Communication Processes in Software Engineering. In Proceedings of the 7th International Conference on Software Engineering (Orlando, FL), pages 358–368. IEEE Computer Society, Los Angeles, CA, March, 1984.

[25] G. Fischer, C. Stevens. Volunteering Information – Enhancing the Communication Capabilities of Knowledge-Based Systems. In H.-J. Bullinger, B. Shackel (editors), Proceedings of INTERACT'87, 2nd IFIP Conference on Human-Computer Interaction (Stuttgart, FRG), pages 965–971. North-Holland, Amsterdam, September, 1987.

[26] B. Fox, L. Karen. Collaborative Cognition. In Proceedings of the 10th Annual Conference of the Cognitive Science Society (Montreal, Canada). Cognitive Science Society, August, 1988.

[27] M.P. Friedman. WANDAH - A Computerized Writer's Aid. Applications of Cognitive Psychology, Problem Solving, Education and Computing. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pages 219–225, Chapter 15.

[28] R. Glaser, K. Raghavan, L. Schauble. Voltaville: A Discovery Environment to Explore the Laws of DC Circuits. In Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), pages 61–66. June, 1988.

[29] E.L. Hutchins, J.D. Hollan, D.A. Norman. Direct Manipulation Interfaces. User Centered System Design, New

Perspectives on Human-Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pages 87–124, Chapter 5.

[30] R. Kass, T. Finin. The Need for User Models in Generating Expert System Explanations. International Journal of Expert Systems 4: 345–375, 1988.

[31] V.E. Kelly. The CRITTER System: Automated Critiquing of Digital Circuit Designs. In Proceedings of the 21st Design Automation Conference, pages 419–425. 1985.

[32] W. Kintsch. The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension. Language Processing in Social Context. North Holland, Amsterdam, 1989, pages 185–209. also published as Technical Report No. 152, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[33] C.P. Langlotz, E.H. Shortliffe. Adapting a Consultation System to Critique User Plans. Int. J. Man-Machine Studies 19: 479–496, 1983.

[34] A.C. Lemke. Design Environments for High-Functionality Computer Systems. PhD thesis, Department of Computer Science, University of Colorado, July, 1989.

[35] C.H. Lewis. Using the 'Thinking-Aloud' Method in Cognitive Interface Design. Technical Report RC 9265, IBM, Yorktown Heights, NY, 1982.

[36] C.H. Lewis, D.A. Norman. Designing for Error. User Centered System Design, New Perspectives on Human-Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pages 411–432, Chapter 20.

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

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

[39] F. Mili. A Framework for a Decision Critic and Advisor. In Proceedings of the 21st Hawaii International Conference on System Sciences, pages 381–386. Jan, 1988.

[40] P. Miller. A Critiquing Approach to Expert Computer Advice: ATTENDING. Pittman, London - Boston, 1984.

[41] P. Miller. Expert Critiquing Systems: Practice-Based Medical Consultation by Computer. Springer-Verlag, New York - Berlin, 1986.

[42] J. Moore. A Reactive Approach to Explanation. Technical Report, USC/Information Sciences Institute, 1988.

[43] 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 SE-11(11): 1337–1351, November, 1985.

[44] J. Psotka, L.D. Massey, S. Mutter. Intelligent Instructional Design. Intelligent Tutoring Systems: Lessons Learned. Lawrence Erlbaum Associates, Hillsdale, NJ, 1988, pages 113–118.

[45] B. Reeves. Finding and Choosing the Right Object in a Large Hardware Store – An Empirical Study of Cooperative Problem Solving among Humans. Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1990. forthcoming.

[46] B.J. Reiser, P. Friedmann, D.Y. Kimberg, M. Ranney. Constructing Explanations from Problem Solving Rules to Guide the Planning of Programs. In Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), pages 222–229. June, 1988.

[47] B.J. Reiser, P. Friedmann, J. Gevins, D.Y. Kimberg, M. Ranney, A. Romero. A Graphical Programming Language Interface for an Intelligent Lisp Tutor. In Human Factors in Computing Systems, CHI'88 Conference Proceedings (Washington, DC), pages 39–44. ACM, New York, May, 1988.

[48] G.D. Rennels. Lecture notes in medical informatics: A computational model of reasoning from the clinical literature. Springer Verlag, 1987.

[49] E. Rich. Building and Exploiting User Models. PhD thesis, Carnegie-Mellon University, 1979.

[50] E. Rich. Users are Individuals: Individualizing User Models. International Journal of Man-Machine Studies 18: 199–214, 1983.

[51] J. Schiff, J. Kandler. Decisionlab: A System Designed for User Coaching in Managerial Decision Support. In Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), pages 154–161. June, 1988.

[52] E. Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. Communications of ACM 29(9): 850–858, September, 1986.

[53] J.F. Sowa. Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley, Reading, MA, 1984.

[54] K. Sparck Jones. Issues in User Modeling for Expert Systems. Artificial Intelligence and its Applications. John Wiley & Sons, New York, 1986, pages 183–195.

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

[56] R.L. Steele. Cell-Based VLSI Design Advice Using Default Reasoning. In Proceedings of 3rd Annual Rocky Mountain Conference on AI, pages 66–74. Rocky Mountain Society for Artificial Intelligence, Denver, CO, 1988.

[57] M.J. Stefik. The Next Knowledge Medium. AI Magazine 7(1): 34–46, Spring, 1986.

[58] W.R. Swartout. Explaining and Justifying Expert Consulting Programs. In A. Drinan (editor), Proceedings of the Seventh International Joint Conference on Artificial Intelligence, pages 815–822. 1981.

[59] R.L. Teach, E.H. Shortliffe. An Analysis of Physicians' Attitudes. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project. Addison-Wesley Publishing Company, Reading, MA, 1984, pages 635–652, Chapter 34.

[60] E. Turban, P.R. Watkins. Integrating Expert Systems and Decision Support Systems. MIS Quarterly: 120–136, June, 1986.

[61] J.H. Walker. Document Examiner: Delivery Interface for

Hypertext Documents. In Hypertext'87 Papers, pages 307–323. University of North Carolina, Chapel Hill, NC, November, 1987.

[62] J.H. Walker. Supporting Document Development with Concordia. IEEE Computer 21(1): 48–59, January, 1988.

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

[64] K. Wipond, M. Jones. Curriculum and Knowledge Representation in a Knowledge-Based System for Curriculum Development. In Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), pages 97–102. June, 1988.

[65] D.D. Woods. Cognitive Technologies: The Design of Joint Human-Machine Cognitive Systems. AI Magazine 6(4): 86–92, Winter, 1986.