University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303)492-7514

User Modelling in Critics
Based on a Study of Human Experts

Gerhard Fischer, Thomas Mastaglio, John Rieman
Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado
Boulder, CO 80309

In *Proceedings of the 4th Annual Rocky Mountain Conference on Artificial Intelligence (RMCAI)*, Denver CO, June 1989

User Modelling in Critics Based on a Study of Human Experts

Gerhard Fischer, Thomas Mastaglio, John Rieman

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado
Boulder, CO 80309

ABSTRACT

Computer-based critics are an effective approach for using knowledge-based systems to support cooperative problem solving but need to be extended with user modelling capabilities. Efforts to do this in the LISP-CRITIC system using statistical methods indicated the need to pursue additional techniques for implicit acquisition of knowledge about the user. A verbal protocol study of human experts analyzing the work of other programmers was conducted. The study focused on how these experts infer the knowledge and expertise levels of anonymous programmers when provided only with samples of the programmers' LISP code. Three distinct categories of "cues" to a programmer's knowledge were found: syntactic, code semantic, and problem semantic. Analysis of these categories indicates that the first two are amenable to acquisition by the LISP-CRITIC system, but that the third category requires a significantly different knowledge base than the system currently contains. Knowledge of the world in general and specific problem domains is required. The results of current efforts to incorporate some of these techniques into the LISP-CRITIC are presented.

KEYWORDS: Computer-based critics, knowledge-based systems, LISP-Critic, cooperative problem solving, models of the user, explanation.

INTRODUCTION

The motivation for this study came from attempts to implicitly acquire information about a user's knowledge of LISP for LISP-CRITIC, a knowledge-based system that critiques LISP code (Fischer, 1987). The goal is a user model that will guide both the critiquing process and the generation of explanations in terms of the underlying concepts of LISP (Fischer, Lemke, Nieper-Lemke, 1988).

Four techniques are available for a computer system to acquire knowledge about the user: explicit questions, testing, tracking tutorial episodes, and implicit approaches. In implicit approaches the system observes the user and makes inferences regarding his or her expertise; this will be the focus of the work discussed in this paper. Because LISP-CRITIC can only use the user's code as a source for this implicit knowledge acquisition, a study was designed to determine the cues that human LISP experts look for when evaluating the code of anonymous student programmers. Our strategy was to analyze the cues humans look for in order to determine which of them could be found through computer analysis of that same code.

THE IMPORTANCE OF USER MODELS IN THE CRITIC PARADIGM

To establish an appropriate context for the discussion we will provide a short description of the target system, LISP-CRITIC, in which we intend to use the results. Our discussion includes an overview of the critiquing paradigm, user modelling in this class of systems, and the specific system itself. For a more detailed description of LISP-CRITIC see (Fischer, 1987, Fischer, Mastaglio, 1989).

The Critiquing Paradigm. *Computer-based critics* are an effective approach for using knowledge-based systems to support cooperative problem solving. Compared with tutoring programs such as LISP TUTOR (Anderson, Reiser, 1985), or PROUST (Johnson, Soloway, 1984), critics do not have knowledge of specific problem domains. Neither do they contain preset examples of the appropriate code for specific problems nor predefined solution paths. Instead, critics allow the user to select the problem domain and goals, interrupting with suggestions when they identify an approach that they prefer to the user's. Critics are applicable in situations where users have some basic competence and are able to at least generate a partial plan or product. They are especially useful in domains where no unique best solution exists but trade-offs must be balanced.

User Models in Critics. Our previous work on critics indicated that they need to be extended with a user modelling capability (Fischer, Mastaglio, 1989). We are interested in incorporating approaches to the user modelling problem attempted in other knowledge-based systems (Clancey, 1986, Kass, Finin, 1987, Fain-Lehman, Carbonell, 1987, Reiser, Anderson, Farrell, 1985, Wahlster, Kobsa, 1988, VanLehn, 1988). Nevertheless, the requirements and constraints on user modelling in critics are sufficiently unique to warrant examining new theoretical ideas.

Unlike tutorial systems, which can track a user's expertise over a path of instruction, computer-based critics must work with users having a variety of background experiences. To operate effectively critics must acquire an

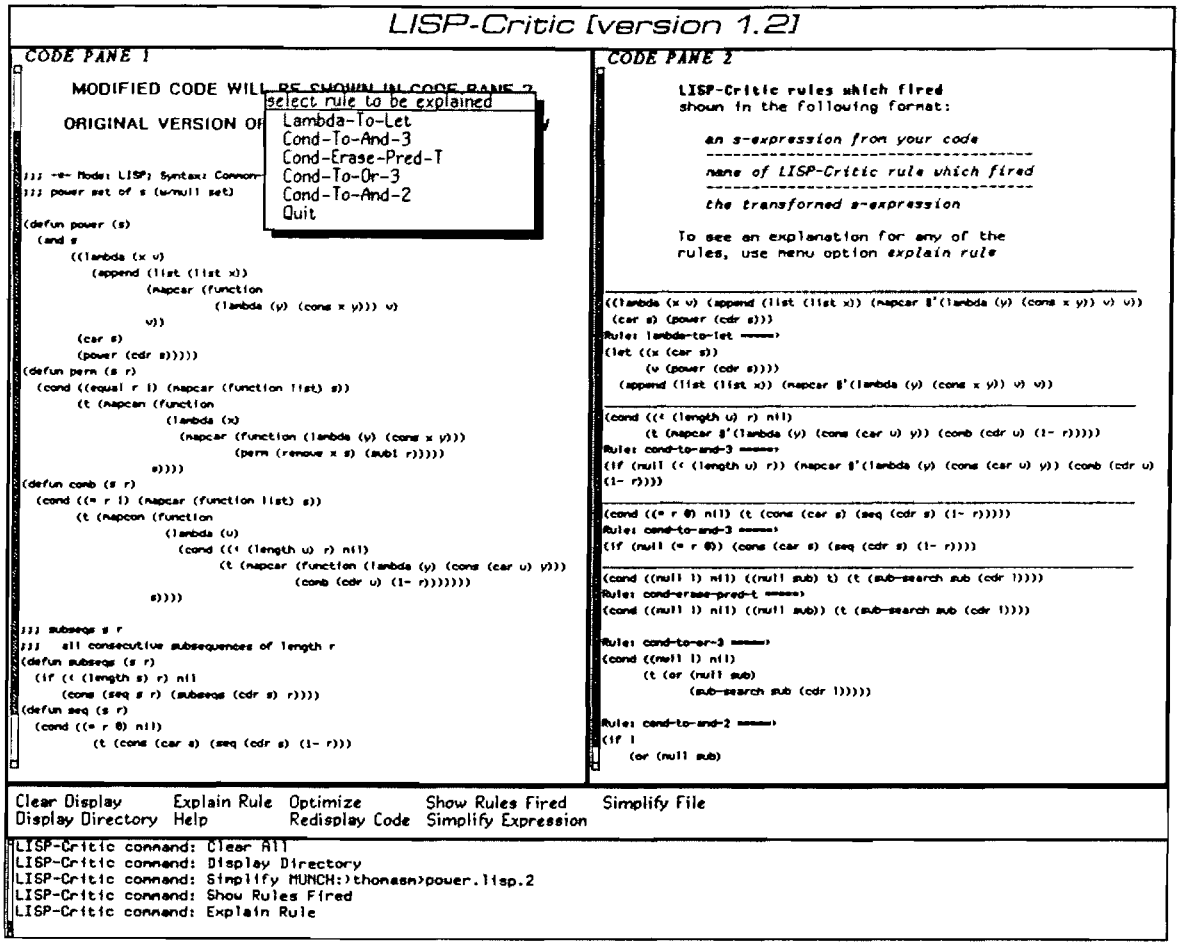

Figure 1: The User Interface of LISP-CRITIC

The interface shows the user working with LISP-CRITIC. The user has seen the recommendations of LISP-CRITIC followed by a rule tracing explanation of the suggested transformations. He is asking for more a more detailed explanation of the rules which were used to generate those transformations.

individual, persistent model of each user. This model, which includes the individual's strengths, weaknesses, and goals, is used in determining (a) the level of performance to expect from the user (and, therefore, at what level to criticize), (b) how to customize explanations, and (c) a basis for tutoring new concepts.

LISP-CRITIC. LISP-CRITIC is an example of a computer-based critic. A user submits working LISP code to LISP-CRITIC, and the system suggests transformations to make the code, at the user's option, more *cognitively* efficient (i.e., more readable) or more *machine* efficient (i.e., faster or requiring less memory). The user's view of the system in operation is shown in Figure 1.

LISP-CRITIC's knowledge base currently consists of over 200 transformation rules, like those shown in Figure 2. The left side of each rule represents a pattern of code, which the system searches for in the user's program. The right side of the rule represents code preferred by the critic which causes the same result. A slot within the rule indicates whether the transformation is for machine efficiency, readability, or both.

It is instructive to compare LISP-CRITIC to the work on the PROGRAMMER'S APPRENTICE SYSTEM (Waters, 1985). The Programmer's Apprentice project attempted to captured knowledge about programming at the plan and goal level. Its knowledge base consists of a set of templates that are appropriate implementations of possible plans. It uses this knowledge base to provide programmers contextually pertinent advice on their work during program design and development. The knowledge in LISP-CRITIC is finer grained in that it knows what constitutes efficient and readable coding style. This knowledge is used to aid programmers in improving LISP code that has been developed to

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) char)
==>
(nconc (explode word) char)
```

Figure 2: An Example Rule in LISP-CRITIC

implement the programmer's plan. A complete programming environment would probably combine the two approaches and contain both types of knowledge.

LISP-CRITIC has proven to be a useful system for many groups of LISP users (Fischer, 1987). However, our formal experiments have show that the user-selectable modes are insufficient to adequately address the needs of a range of user expertise. A user model containing information about the user's abilities and goals could be used by the system as follows:

- *To determine what subsets of rules to fire for each individual.* For example, a novice user with the goal of learning to write machine efficient programs would probably not have the background knowledge to benefit from a rule that transformed a segment of code into a macro.
- *To customize explanations so they cover exactly what the user needs to know.* A user who already understands the distinction between destructive and copying functions, but uses *append* where *nconc* could be applied, may be satisfied with the simple explanation that "*nconc* is a destructive function and *append* is a copying function".
- *To provide the information needed to place the user within a temporary tutoring environment.* Tutoring episodes can play an important role within the critic paradigm -- the crucial difference from the normal tutoring approach is that tutoring suggested by the critic takes place in the context of the user's work.

Acquiring Knowledge About the User Within the Critic Paradigm: Several general techniques can be used to acquire knowledge about the user.

- Explicit: The system may ask the user questions concerning general background ("How many years have you been programming?") or specific concepts ("Do you know how to use mapcar?").
- Tutorial: The system may lead the user through a graduated series of learning exercises, evaluating performance and knowledge in the process. The user model can be updated to show that the user has been instructed on certain concepts and should understand them.
- Test: The system, even if not used in a tutoring mode, may pose specific problems for users as a means of acquiring knowledge about them.
- Implicit: The system may infer knowledge about users from their actions during normal system operation. A system module has been developed that performs statistical analysis of the submitted file, yielding data on what functions and constructs the users have employed (Fischer, 1987). Because the critic paradigm emphasizes user control and minimizes intrusion, the implicit acquisition of knowledge was deemed worth additional research. We decided to observe the techniques that human LISP experts use when presented with some code and asked to evaluate the code writers' knowledge. It was hoped that some of the techniques could also be used by LISP-CRITIC.

METHODOLOGY

Data. Programs completed and turned in as homework in an undergraduate "Introduction to Artificial Intelligence" class were collected. The student programmers were college juniors and seniors, primarily computer science majors; all had previous experience with at least one other programming language. This was the first formal instruction most had received on LISP. The programs of six students, chosen to provide a variety of solutions, were used. The programs were the students' solutions to the first two course assignments in LISP. The assignments required nine different problems to be solved, some of which, depending

on the strategy selected, involved defining several functions. The individual functions varied from 4 to 39 lines in length and 10 to 66 s-expressions in size, depending on the difficulty of the problem and the individual programmer's style.

Subjects. Seven expert LISP programmers were used as subjects. They included professional AI Researchers and graduate students with extensive experience in LISP. Some of the subjects have taught programming in general as well as LISP in particular. With one exception, the subjects did not know the students who had written the code.

Procedure. The human experts were asked to examine and analyze the code to determine the expertise of the student programmer using a talking protocol methodology (Ericsson, Simon, 1984). Subjects spent from 1 to 2 hours performing the analysis. Not every expert analyzed all of the code of each programmer; this was acceptable because the study attempted to identify cues each expert focused on rather than correlating the experts analyses.

Some of the subjects were content to examine the code at the level of programming language syntax, not caring how or even if the programs worked. Others were unable to make judgments until they had traced the code execution and convinced themselves of how specific functions operated, even though they were told that this was not necessary.

LIMITATIONS OF THE APPROACH

Fundamental Limitations. At the outset, certain limitations of this approach were recognized. First, in attempting to evaluate a programmer's knowledge by examining the individual's code, an expert can only make inferences concerning programming techniques and concepts that are used (or clearly should have been used) in the problem at hand. Similarly, a single correct or incorrect use of a technique is not conclusive evidence of a programmer's level of knowledge of the underlying concept. Computer systems as well as humans must deal with both these situations.

Methodological Limitations. In addition to these fundamental limitations, there were certain restrictions in the methodology used. No attempt was made to determine what use the experts made of the cues they found, nor to correlate the experts' judgments with an independent metric of the programmers' ability or code quality. The focus was on the types of cues experts noted in other programmers' code, not the method used or ability of experts to evaluate those cues.

The problems solved by the programmers were simple, the techniques known by the novice programmers were limited, and the number of experts who analyzed the code was small. It was not expected that the study would yield a comprehensive list of cues that could be found in LISP code. We wanted to gain insight into the characteristics of the cues that experts used.

Finally, it is apparent, in retrospect, that the results might have been more applicable to LISP-CRITIC if the types of information not available to LISP-CRITIC had been masked from our experts -- problem specific information such as meaningful variable names, comments, and problem description. The fact that our experts were shown the requirements specified by the course instructor created a situation dissimilar to LISP-CRITIC. LISP-CRITIC does not attempt to capture the specific problem description and perhaps our experts would have made slightly different inferences if they had been insulated from this information.

DISCUSSION OF PROTOTYPICAL EXAMPLES

In Figures 3 and 4 we show excerpts of code from two student programmers and a summary of the comments from four experts. The assignment was to write a function called *powerset* that takes a list and returns a list of all possible combinations of lists within the input list. The experts

```
;*****************************************************************
;
; Function:    ps list
; Variables:   list, list of elements that the power set is desired
; Returns:     list of list, the power set of the elements in the list
; Side Effect: none
;
;*****************************************************************
 (defun ps (l1)
   (cond ((null l1) (list l1))
         (t (let ((l2  (ps (cdr l1)) )
                  (pre (car l1) ))
                  (append l2 (mapcar #'(lambda (l) (cons pre l))l2))
                 )) ))
```

Figure 3: LISP Code Submitted by First Student.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; 2/7/88
; PS takes a list 'lst' and returns the powerset of that list.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun ps (lst)
       (cond ((null lst) (cons lst lst))
           (t (append (ps (cdr lst)) (mapcar (function (lambda (elt)
                                                 (cons (car lst) elt)))
                                             (ps (cdr lst))))
              )
       )
 )
```

Figure 4: LISP Code Submitted by Second Student:

were shown the written assignment and then asked to verbally react to the code, including comments, orienting on how they go about would evaluating the knowledge of each programmer.

First Student's Code. The subjects made the following comments on the code in Figure 3 (italics and bracketed material added to identify specific cues):

Expert 1: could use *destructive function* instead of mapcar; let instead of setq is good [*lexical scoping*]; should use "powerset" instead of "ps" [*semantically meaningful names*].

Expert 2: doesn't understand the let statement... wait, maybe he does... yes, he does, but he's *indented* it wrong.

Expert 3: puts carriage return between this function and previous one, that's good [*formatting*]; very good because lambda used inside mapcar [*higher level functions, lexical scoping*]; doesn't *indent* the let properly; I would have used *if rather than cond*, it uses less parentheses.

Expert 4: uses mapcar and let, wow! [*higher level functions*]; the cond could be simplified, but sometimes that is silly [*cond-to-if*]

Second Student's Code. The subjects' comments on the program in Figure 4:

Expert 1: uses the function [ps] twice [*minimal code not used*]; *does not use destructive functions.*

Expert 2: could change *cond to if*; note mapcar [*higher level function*] and *recursion.*

Expert 3: calls powerset of (cdr lst) twice [*minimal code not used*]; *indentation* is bad, because he doesn't know where to put the carriage return, and he doesn't conserve space where he can.

Expert 4: doesn't cache the car of lst and doesn't cache the powerset of the rest of the list, which will increase computation time [*caching*].

ANALYSIS OF RESULTS

Collectively the subjects mentioned 67 different cues in their protocols. The cues listed in Figure 5 were mentioned by three or more subjects. Our analysis indicated that the cues fell into three categories:

1. Cues that are a matter of looking at the syntax of the code and inferring from the functions used that the programmer possesses or lacks certain knowledge. An example of demonstrated knowledge is the use of mapping functions; an example of lack of knowledge is the use of copying functions when destructive functions would achieve the same result.

2. Cues that result from examining deeper syntactic and semantic structure of the code, generally requiring some understanding of how the code is manipulating data structures. Examples here are use of recursion or lexical scoping.

3. Cues that result from examining the overall program structure and/or problem semantics. Semantic variable names and use of a structured programming language style (similar to C or Pascal), are examples in this category.

IMPLICATION FOR COMPUTER ACQUISITION OF THE USER MODEL

The information obtainable for the least computational cost are the cues found by identifying the functions used. The positive versions of these can be identified through the statistical code analysis module already completed (Fischer, 1987). The negative versions of the cues -- that is, use of a given function when a more efficient function is possible -- can be inferred by recording which LISP-CRITIC rules have fired.

CUE	CATEGORY
Copying instead of destructive functions used	1
Macros not used when possible	1
Higher level functions used	1
Correct equality test	1
Mapping functions used	1
Recursion used	2
Lexical scoping utilized	2
Similar clauses inside a conditional	2
Structured Programming Language coding style	2
Extraneous (unnecessary) function calls	2
Error conditions checked	2
Use of specialized data structures	2
Proper indentation	2
Minimal code used	3
Variables named semantically	3
Meaningful comments	3
Problem decomposed properly	3

Figure 5: Cues Mentioned by Three or More Subjects

The cues in the second category, which require examination of the code's structure, can be identified by additional preprocessing or postprocessing of the user's code. Use of recursion is a cue in this category. A preprocessor can identify recursion by searching for calls to a function within the body of the function's own definition.

Cues in the third category are beyond our approach and will be very difficult for any computer system to obtain. These cues, which result from examining program meta structure and/or problem semantics, require experts to use knowledge in addition to what they know about LISP. Knowledge required is of the general world type, for example, recognizing that a variable name has semantic meaning that represents a real world entity or concept even when it is spelled phonetically.

A PROTOTYPE USER MODEL FOR LISP-CRITIC

The insights gained from observing experts were used as a basis for a prototype user model for LISP-CRITIC. An overview of the implementation is shown in Figure 6. Acquisition of the user model begins with cues, which the system looks for in the code of the user. Certain combinations of cues cause the system to assume that the user knows a particular concept. For example, a user who makes a recursive function call and also correctly uses a conditional to terminate the recursion is assumed to know recursion. The user model consists of a number of chunks and is updated with each use of LISP-CRITIC. A confidence value that is determined by the source of the knowledge and a knowledge value indicating the user's skill are assigned to each chunk.

The information accumulated in the user model is combined with knowledge about stereotypes [Rich 79; Rich 83]. When certain combinations of chunks reach a threshold value, a particular LISP programmer stereotype would trigger, and the values for *all* the chunks within that stereotype are incremented.

The user model also includes user preferences or knowledge about related domains. For example, a series of cues may cause the model to record a user's preference for if-then-else and loop structures, and this can trigger a stereotype that increments values of other chunks related to programming concepts common to procedural languages. Such information is useful for tutoring and explanation. A user with experience in other programming languages could understand an explanation of the difference between *nconc* and *append* in terms of pointers; a user with no programming experience would require a different type of explanation, for example, a visualization of the underlying structures using KAESTLE (Fischer, 1987).

The prototype implementation we have described makes use of the same techniques used by human experts, but many problems remain. Some of these are what the user model should include, how to structure that knowledge, and how to define stereotypes. The last item proved to be a hurdle which we could not easily surmount. It would be nice to report that this model has been fully implemented and proven to be efficacious, however this is not the case. Our efforts to determine the stereotypes that the system needs to function in the manner described became a fundamental limitation. There is not an agreed upon set of

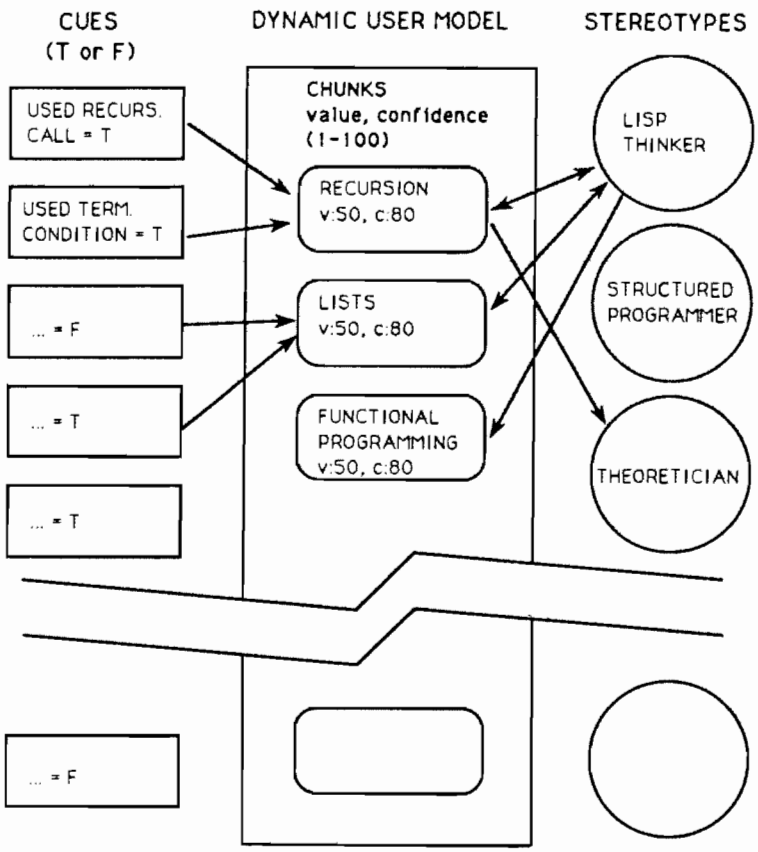

Figure 6: User Model and Knowledge Acquisition for LISP-CRITIC

The left side of the model shows *cues* which LISP-CRITIC looks for in the user's code. When appropriate cues are found, a *chunk* of knowledge within the user model is assigned a certain value (v) and confidence level (c). After several appropriate chunks are found, a stereotype fires, assigning confidence values to other chunks not inferred by direct observation by LISP-CRITIC. Chunks in the latter category were not previously been observed by LISP-CRITIC in the user's code.

LISP programmer stereotypes which one can determine by consulting references on LISP or a group of LISP experts. The alternative is to perform statistical analysis of a large collection of LISP code from a wide population of users. This was determined to be not only a difficult task but because the statistical results still require interpretation, one whose validity is not necessarily guaranteed.

Our current approach to implicitly acquiring the user model is to incorporate the techniques observed in our experts as part of a larger collection of such techniques. These infer the user's understanding of the underlying concepts and specific functions of LISP. The user modelling component consists of a user database and a collection of modelling agents as shown in Figure 7. The cues that our experts used to infer the user's knowledge state will be incorporated in rule form into the modelling agent as a technique for updating the user model itself. Code analysis provides the information that will trigger these rules.

SUMMARY

Our protocol study was oriented toward identifying the techniques used by human experts in analyzing the work of others and then using these techniques together with known stereotypes of users in the application domain. We met fundamental limitations in applying the results in this manner, but are still attempting to integrate what we learned about human behavior into a computer system tasked to perform a similar task. This is a form of cognitive modelling that does not attempt to achieve a veridical model of the human processes but instead seeks an efficacious methodology by which the machine can perform a similar function.

The critic paradigm is a powerful approach to cooperative problem solving in which a user model can play a role. It can be used to guide criticism, to customize explanations, and to provide the information for contextual tutoring. The

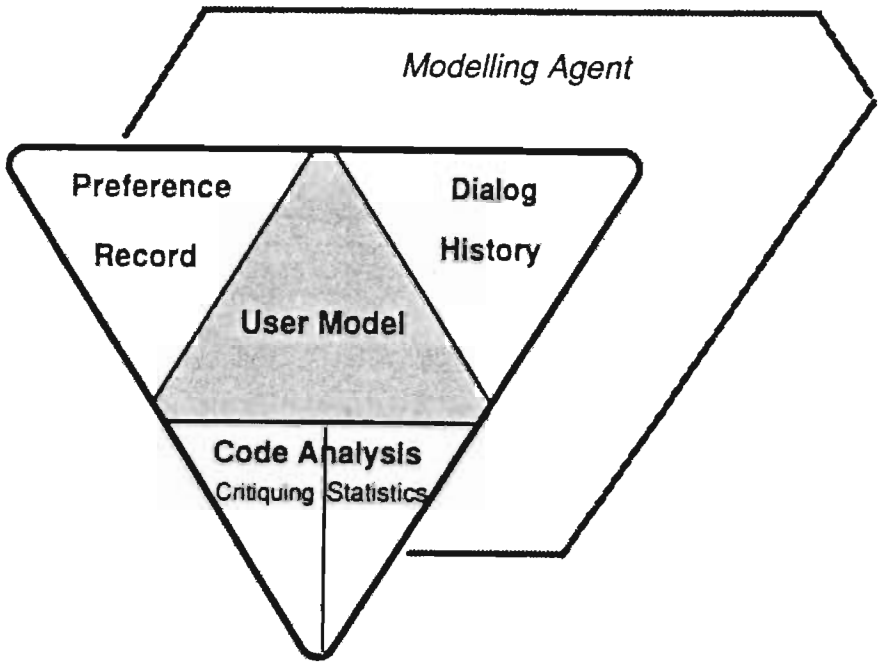

Figure 7: The User Database

research described in this paper focused on acquiring a user model for LISP-CRITIC, a system that suggests transformations to improve LISP code. It was found that when human experts analyze the code of other programmers, they look for certain cues, from which they make inferences about the programmer's knowledge and ability. Many of the same cues can be extracted from the code with tools that have been developed in the context of LISP-CRITIC research. A large subset of the cues, however, depends on problem-specific and general world knowledge that a computer-based critic does not have. A simple user model has been created that shows the value of the easily identifiable cues. Research is continuing into how knowledge should be represented within the user model, how the accuracy of that knowledge should be verified, and how the model can be used by other critic system components.

ACKNOWLEDGMENTS

We would like to acknowledge the participation of the seven LISP experts who voluntarily served as our subjects: Jonathan Bein, Brigham Bell, Bernard Bernstein, Hal Eden, Andreas Lemke, Helga Nieper-Lemke, and Christian Rathke. Paul Johl assisted in collecting the protocols and contributed to the prototype model for acquisition that uses stereotypes. We also appreciate the cooperation of Brigham Bell for assisting us in collecting the program samples from his students and, of course, the six anonymous students whose code was used in this experiment.

REFERENCES

J.R. Anderson, B.J. Reiser, "The LISP Tutor", *BYTE*, Vol. 10, No. 4, April, 1985, pp. 159-175.

W.J. Clancey, "Qualitative Student Models", *Annual Review of Computing Science*, Vol. 1, 1986, pp. 381-450.

K.A. Ericsson, H.A. Simon, *Protocol Analysis: Verbal Reports as Data*, The MIT Press, Cambridge, MA, 1984.

J. Fain Lehman, J. G. Carbonell, "Learning the User's Language: A Step Toward Automoated Creation of User Models", Tech. report, Carnegie-Mellon University, 13 March 1987.

G. Fischer, "A Critic for LISP", *Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy)*, J. McDermott,ed., Morgan Kaufmann Publishers, Los Altos, CA, August 1987, pp. 177-184.

G. Fischer, A.C. Lemke, H. Nieper-Lemke, "Enhancing Incremental Learning Processes with Knowledge-Based Systems (Final Project Report)", Tech. report CU-CS-392-88, Department of Computer Science, University of Colorado, March 1988.

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

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.

R. Kass, T. Finin, "Modelling the User in Natural Language Systems", *Computational Linguistics, Special Issue on User Modeling*, Vol. 14, 1987, pp. 5-22.

B.J. Reiser, J.R. Anderson, R.G. Farrell, "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, AAAI, 1985, pp. 8-14.

E. Rich, *Building and Exploiting User Models*, PhD dissertation, Carnegie-Mellon University, 1979.

E.A. Rich, "Programs as Data for their Help Systems", Tech. report, University of Texas, Austin, 1983.

K. VanLehn, *Toward a Theory of Impasse-Driven Learning*, Springer-Verlag, New York, 1988, pp. 19-41, ch. 2.

W. Wahlster, A. Kobsa, "User Models in Dialog Systems", Tech. report 28, Universitaet des Saarlandes, FB 10 Informatik IV, Sonderforschungsbereich 314, 1988.

R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November, 1985, pp. 1296-1320.