Science and Technology, Special Review (2000) 157-183
© 2000 Sultan Qaboos University

Intelligent Systems for Active Program Diagnosis

Haider Ali Ramadhan and Khalil Shihab

Department of Computer Science, College of Science, Sultan Qaboos University, P. O. Box 36, Al Khod 123, Muscat, Sultanate of Oman.

أنظمة البرمجه الذكية للتحليل الفوري

حيدر علي رمضان و خليل شهاب

خلاصة : يتناول هذا البحث دراسة تحليلية و مقارنة لانظمة البرمجة الذكية القادرة على اكتشاف الاخطاء المنطقية في البرنامج. كما يتناول البحث طبيعة المعالجة الذكية للبرنامج في هذه الانظمة, و الوسائل المستخدمة لتمثيل المعلومات لضرورية لعملية المعالجة. كما يطرح البحث السبل البديلة للتغلب على السلبيات الموجودة في الانظمة الحالية.

ABSTRACT: Intelligent program diagnosis systems are computer programs capable of analyzing logical and design-level errors and misconceptions in programs. Upon discovering the errors, these systems provide intelligent feedback and thus guide the users in the problem-solving process. Intelligent program diagnosis systems are classified by their primary means of program analysis. The most distinct split is between those systems that are unable to analyze partial code segments as they are provided by the user and must wait until the entire solution code is completed before attempting any diagnosis, and those that are capable of analyzing partial solutions and providing proper guidance whenever an error or misconception is encountered. This paper gives an overview of the field and then critically compares work accomplished on several closely related active diagnosis systems, emphasizing such issues as the representation techniques used to capture the domain knowledge required for the diagnosis, ability to handle the diagnosis of partial code segments of the solutions, features of the user interfaces, and methodologies used in conducting the diagnosis process. Finally the paper presents a detailed discussion on issues related to active program diagnosis along with various design considerations to improve the engineering of this approach to intelligent diagnosis. The discussion presented in this paper tackles the issues referred above within the context of DISCOVER, an intelligent system for programming by discovery.

KEYWORDS: Intelligent programming systems, knowledge representation, program debugging, software development, user programming.

CONTENTS

1. Introduction : Automatic Program Debugging

1.1 The Background

Over the past twenty years, the area of user programming has undergone a major change in perspective and direction. In the beginning, the emphasis was centered on the needs of the users in trying to learn programming and problem-solving. Intelligent programming systems developed for this purpose targeted the *design and the planning* sides of programming, which embodied an *instruction-oriented* paradigm. Users were led through a structured manner, embarking on well-defined tasks, and even designing and implementing their programs according to a model of *good practice or ideal solution*. This model was critical because it facilitated automated diagnosis when bugs were introduced by the users. This approach of *instruction-oriented* paradigm appeared to embody a clean, top-down, and reliable software engineering practice, since it aimed to get the design and specification error-free from the beginning. Unfortunately, in

reality, it also acted as a straight-jacket in many situations for novice users. Examples of these systems include (Bonar, 1992; Anderson, 1990; Reiser, 1992; Murray, 1986; Johnson, 1980).

Subsequently and due to limitations associated with the above approach to automatic debugging, and in general with the difficulty in coming up with automatic debuggers which would cater for more than small and toy programs, the efforts in this area shifted to the *software maintenance* side of user programming. Discovery programming environments along with support for visual techniques were the focal point of this new direction. Efforts invested in this direction seemed to claim that helping the users to understand the dynamic behavior of programs and algorithms during execution could be proven to be more effective in teaching programming to novice users. The result of this direction created a shift from automated debuggers to software visualization.

Ever since, much of research in user programming has been polarized toward these two opposite domains: intelligent programming systems and discovery programming environments. The intelligent systems concentrated mainly on helping novices in acquiring programming skills through a series of problem solving situations. Most of these systems ignored the significance of incorporating visualization and discovery features which would also help the users in compiling effective programming knowledge. Discovery systems, on the other hand, concentrated mainly on helping the users in building correct programming knowledge through visual and discovery environments. These systems neglected the issue of supporting intelligent diagnosis and tutoring through which novices can transform their knowledge into programming skill. Examples of these systems include Agentsheets (Repenning, 1996), SEE (Baecker, 1990), TANGO (Stasko, 1992), ZEUS (Brown, 1992), and TPM (Eisenstadt, 1993).

Recently, another improvement on the general lines discussed above has emerged. This recent direction emphasizes the incorporation of visualization and discovery features into intelligent program debuggers in order to come up with guided programming environments, which may hopefully help novice users in building both the programming knowledge and the problem-solving skill. A solid practical example of such recent direction is exhibited by the DISCOVER system (Ramadhan, 1992a, 1992b, 1997, 1998, 1999a, 1999b, 2000a, 2000b). Empirical evaluations of the system seem to support the usefulness of this direction.

With the advancement in the Internet computing and Web development, teaching user programming over the Web is becoming a very promising direction. World Societies will become more reliant on proficiency in programming, so that teaching it quickly and effectively will become more important. With the growth of distance learning technology, access to learning systems is expected to grow, and the lessons learnt from teaching programming have the potential to be applied to other areas such as medical diagnosis and circuit design and layout.

Among the three general programming paradigms mentioned in the above discussion, this paper deals with only two types of systems: (1) automatic program debuggers, and (2) guided discovery programming systems. Discovery programming environments with no support for automatic program diagnosis are not covered in our analysis. It is important to note here that due to difficulty encountered in the development of systems for automatic program diagnosis, very little progress has been reported in this interesting area especially in recent years. It is for this reason that most of the systems mentioned in this paper are the ones developed some considerable time ago. However, it is the methodology embodied in these systems that constitutes the core of this paper and not the systems themselves. We have also put considerable effort in suggesting better design directions to improve the engineering of new systems for program diagnosis.

1.2 A general Model of Program Debugging

Before describing specific systems that support automatic program diagnosis, it is worth looking at a general model of program diagnosis. This will facilitate characterizing and comparing various approaches and systems that have been developed in this area.

An automatic program diagnosis and debugging system is a computer based program capable of analyzing programming solutions and providing intelligent feedback, and thus supporting problem-solving

domains. The program diagnosis methodologies discussed in this paper are basically specializations of the following general model. Task specifications are mapped to the code provided by the user. This mapping process allows the task of program debugging to be decomposed into smaller and simpler steps. When decomposition is no longer possible, the diagnostic critic looks for discrepancies between code and specifications and interprets these as bugs. When intelligent tutoring is also supported by the system, this information from the critic is passed over to the tutorial expert that interacts with the user model and provides tutorial interactions and instruction.

The task specifications are precise criteria for successful completion of the debugging task. The following methods of task specifications have been used by the automatic program debugging systems indicated:

- Model-answer programs (reference solutions): Laura (Adam, 1980), Aurac (Hashmer, 1983), Talus, Bridge, and DISCOVER.
- Input/output pairs: BIP (Barr, 1976).
- Goals to be achieved (specified in some special language or format): Proust, Bridge, GREATERP, GIL, Aurac, MYCROFT (Goldsein, 1974), and DISCOVER.
- The expected trace: PDS6 (Shapiro, 1983).
- Constraints on program output: MYCROFT.

The diagnostic critic compares task specifications and program code for discrepancies that can be interpreted as bugs. The following indications have been used to flag the detection of bugs:

- Mismatch between plan templates and student code: Proust, DISCOVER.
- Inability to synthesize student code (simulate the ideal student): GREATERP, GIL.
- Differences between expected and actual execution traces: BIP, PDS6.
- Inability to verify program specifications: Talus, Aurac.
- Inability to satisfy a list of program requirements: Bridge.
- Violations of output constraints: MYCROFT.

2. Active Versus Passive Debugging

Automatic program diagnosis and debugging systems can be classified by their primary means of program analysis. The most distinctive split is between those systems that are unable to analyze partial code segments as they are provided by the user and must wait until the entire solution code is completed before attempting any diagnosis, and those that are capable of analyzing partial solutions. The former perform *post-event analysis* while the later perform *in-event analysis*.

Systems using post-event analysis can be further divided according to their methods of isolating and localizing errors into (1) those using specification based analysis, such as Proust, Pudsy (Lukey, 1980) and Aurac, (2) those using trace-based analysis, such as PDS6, (3) those using I/O based analysis, such as BIP, and (4) those using model-answer based analysis, such as Laura and Talus.

Systems using in-event analysis can be further divided according to their methods of reasoning about the user into those supporting active analysis and those supporting passive analysis. Systems using passive analysis do not trace the intentions of the user or his design decisions while being developed and require him to explicitly request the automatic debugging of his code segments. These systems localize errors in the user programs either by looking for surface structural forms (plans) (Rich, 1986) or by accounting for differences between forms and actual code segments, as in the case of Bridge. Generally speaking, these systems rely on some sort of pre-stored requirements for a complete solution, and hence are classified in our taxonomy under model-answer based systems. It is worth noting here that systems which rely on pre-

stored requirements for a successful solution cannot solve the problems themselves, and hence cannot reason about the solutions and designs provided by the users (see section 2.2).

On the other hand, systems using active analysis perform automatic debugging by implementing model-tracing (Anderson, 1990). Through this approach, these systems subdivide tasks into smaller steps that must be solved one at a time. The user's design decisions are traced as he develops the solution. During each step taken by the user, these systems check to see if the user is following a design path known to be correct or buggy. Buggy paths are pruned as soon as they are detected by giving the user intelligent feedback and allowing him to try again. Examples of such systems include GREATERP, GIL and DISCOVER. These systems tend to be quite directive. However, through rich interaction and flexible immediate feedback, these systems detect very specific bugs and misconceptions.

There is some overlap in these categories. DISCOVER performs model-tracing, but relies on a pre-stored model answer (the reference solution) to represent its knowledge about the ideal user. Aurac performs specification based analysis to match code segments with the library of program clichés, but then later on uses its model-answer algorithm to conduct data-flow analysis.

2.1 Advantages of Active Debugging

Several advantages of the active approach to automatic program debugging can be outlined as follows:

- Very specific errors can be diagnosed and feedback can be given in proper context of the error, hence the users can be explicitly guided in the process of acquiring problem-solving skills.
- Systems using this approach are capable of analyzing partial solutions as they are provided by the user and therefore have access to all intermediate states, hence they work with more information than systems that are capable of only analyzing complete solutions. This in turn provides these systems with the capability of reasoning about the programming process itself and thus generates very specific explanations and advice.
- The impact of multiple bugs on the diagnosis process is minimized. Many of the post-event based systems such as Proust, Talus, Aurac and Laura have to deal with disentangling multiple bugs which require them to generate all possible alternative treatments of these bugs and pick the best from among them. Systems using active analysis simply prevent the user from making multiple bugs and explain each bug immediately. Therefore, the code never contains more than one bug at a time. In addition to cognitive justifications, (see Anderson, 1982), this approach greatly simplifies the engineering and the implementation of automatic program diagnosis.

Despite these advantages, this approach to automatic program diagnosis tends to be very directive. In addition, such approach ignores the issue of providing the user with some chance to detect and correct bugs on his own, since the user is not allowed to go wrong. However, these disadvantages can be overcome by supporting a more flexible style of user interaction while still retaining close ties to model-tracing. This was successfully accomplished by the DISCOVER system through (1) supporting an ability to give delayed feedback by increasing the grain size of automatic diagnosis to a complete program statement, not just a single word or symbol, and (2) allowing the user to do limited backtracking by giving him some chance to delete previously entered code and restart, while in full interaction with the system (see section 4 for more detail).

2.2 The Implementation of Active Debugging

One way to implement model-tracing, active program debugging is to provide the system with a set of problem solving rules (a production system) that allow it to model the user by generating possible steps that a user might take while solving a given problem. Thus, while the user is working, the system simulates the steps that an ideal user could take in completing the program. In addition, this approach also models possible errors that the user makes at each step on the basis of pre-known bugs and misconceptions stored in a library. By comparing the user's actions (partial solution steps) to the set of possible correct actions and

the set of buggy actions, the system can determine whether the user is moving on a correct solution path or showing evidence of bug. This combination of correct and buggy sets of rules is referred to as the student model, while the process of comparing the user's actions to the ones generated by these rules is referred to as model-tracing. Both GIL and GREATERP follow this model.

DISCOVER followed a different approach in implementing model-tracing. In this approach the system is provided with a pre-stored reference solution that represents the ideal solution, but with no account for possible and common bugs. The system much like Laura, Talus, Aurac and Bridge, applies various heuristics and pattern matching techniques to match steps taken by the user with parts of the reference solution. The system analyzes the surface code provided by the user without much specific knowledge about the problem to be solved or about how to design and construct an algorithm. Therefore, unlike production based systems, systems that rely on pre-stored reference solution cannot solve the problems themselves. Since these systems rely on manipulating the surface code, it is possible to describe to the user *what* the next step in the solution is but it is not possible to reason about *why* the step is appropriate.

However, this problem can be bypassed to some extent by incorporating hand-coded explanations for each reference solution. In fact, by augmenting each complete step in the reference solution with a hand-coded explanation that simulates the reasoning process, it becomes possible for DISCOVER to describe to the user quite specifically why each of his actions is appropriate or not.

3. Survey of Approaches

As described above, automatic program debugging and tutoring systems can be classified by their primary means of program analysis. The most distinctive split is between those debugging systems that are unable to analyze partial code segments as they are provided by the user and must wait until the entire solution code is completed before attempting any automatic debugging or tutoring, and those that are capable of analyzing partial solutions. The former perform *post-event analysis* while the letter perform *in-event analysis*.

3.1 Post-event Systems

Post-event analysis systems are characterized by their inability to diagnose partial, incomplete solution steps. These systems also lack rich interaction with the user and require him to sufficiently understand both the detailed knowledge of the syntax and the semantics of the language constructions (programming knowledge), and the process of relating these constructs along with their semantics to come up with correct programs (algorithmic or programming skill). The remainder of this section examines some of these systems in more details.

3.1.1 Specification-based Analysis

Automatic debugging systems that perform specification-based analysis are provided with a high-level description of the goals of the user's code and they check to see to what extent these goals are satisfied by the user's code. Examples of such systems include Proust, Aurac, Pudsy, and MYCROFT.

Proust uses stored plan templates to match against parsed student code. The program specification is expressed as a sequence of task goals. Proust stores a plan library that associates task goals with plan templates. Plan templates are associated with the expected code in the user's plan that they can match against. Proust infers the intentions of the student program statements by matching each statement to some part of a plan. When all goals are achieved, Proust fully determines the intentions of the student program. When more than one plan matches a statement, Proust uses heuristics to select the one that expects the fewest bugs in the student program. By using heuristics, Proust avoids impractical exhaustive search.

Aurac attempts to match segments of the student program written in Solo against a library of clichés. Because of the simplicity of the Solo domain, Aurac manages to store programming clichés that would match most segments of the student program. Partial matching of the clichés to the code, is treated as a

candidate bug. In the final stage, the system uses data flow analysis to detect such errors as unused bounded variables. Aurac is also capable of recognizing some simple algorithms. By making sure that the high-level goals specified by a model algorithm for a given algorithm are satisfied by the student code, Aurac can determine if the code is logically correct. The system checks each line of the code against sample lines from an algorithm. When all lines of a given algorithm are found, Aurac states that the code is correct.

Pudsy uses a specification as the high-level description of the correct code, and matches the output of the student program against a specification. Pudsy breaks the code into smaller logical chunks. The system then relates each chunk to a particular task in the problem, using a record of tasks associated with the high-level description of the program's goals. In the first pass, Pudsy looks for local clues that suggest bugs in chunks, such as redundant assignments. In the second pass, Pudsy figures out low-level assertions about the values of variables on exiting each chunk and then transforms these low-level assertions into a high-level description. The description is then matched against a specification. On mismatch between specification and description, Pudsy examines the assertions built so far and determines the code segment responsible for discrepancies between specification and description.

MYCROFT examines the side effects produced by a program. The domain is drawing simple pictures in LOGO and the side effects are lines drawn and changes to the turtle state. Like Pudsy, MYCROFT uses a specification as the high-level description of the correct code, and matches the output of the student's code against a specification. The specification describes the relationships between the components of the shapes drawn. On finding a mismatch between the drawing and the specification, MYCROFT determines the bug to be in the code that produced the drawing.

3.1.2 Trace Analysis

Systems that perform trace analysis engage in a debugging dialogue with the user. PDS6 is an example of such a system. The system interactively debugs Prolog programs by monitoring program execution. The system builds a tree of the calling sequence of the procedures involved. It then asks the user questions about the desired and actual behavior of the procedures. By comparing actual program execution with the desired execution, PDS6 can determine the buggy procedure. The system relies heavily on an 'oracle,' typically the user, to answer questions about the expected behavior of the program.

3.1.3 Model-answer Analysis

Debugging systems that perform model-answer based analysis attempt to match pre-stored, possibly parsed, model programs to the parsed student code. Examples of such systems include Laura, Talus and Ruth's system (Ruth, 1973).

Laura analyzes the surface code of the student program. The system transforms the student program written in FORTRAN and the model program into graphs and then normalizes these graphs. Normalization transforms the graphs into a standard form. An example of transformation used might be - if the same variable is used for two different purposes then a new variable is generated. This makes the matching process easier. Any discrepancies discovered between these graphs during the matching process are considered to be bugs.

Talus debugs programs in Lisp by reasoning about the computational semantics of the programs. Recursive programs are compared to model programs. These programs act both as specifications and sources for correcting the buggy code. An inductive proof of equivalence is constructed to compare student and reference programs. Where the proof would fail, the student program is altered with code from the reference program. Talus applies various heuristics to pair reference functions with student functions and to pair formal variables with actual variables. The system simplifies the student code using a sequence of program simplification transformations. This process transforms the programs into IF-normal forms, which then facilitates algorithm recognition and bug detection by reducing the variability of the programs. Talus detects bugs by generating and evaluating verification conditions that are required for the inductive proofs

that establish the functional equivalence of each student function to its paired function. When the proof would fail, Talus knows that a bug has been discovered.

Ruth's system uses 'generative semantic grammars' to represent plans, and student programs are parsed in terms of the grammar. This method is similar to syntactic analysis in natural language processing. The system analyzes simple sorting programs with this approach and a grammar similar to an Augmented Transition Network (ATN). The system uses the grammar to translate a given algorithm into the program submitted by the student. If the translation fails, then the program is considered to be incorrect. The system also uses the same grammar to generate the problems in the first place.

3.2 In-event Analysis

Automatic program debugging systems which diagnose partial solutions as they are provided by the user perform in-event analysis. The systems normally emphasize rich interactions and a visible user interface. These systems allow the user to explore the problem-solving process through an interactive exploration of programming functions, concepts and plans. The user is not required to have extensive programming knowledge and programming skill to be able to explore programming in these systems. The user learns by trying out his hypotheses which are represented by the fragments of knowledge he might have about programming, and it is up to the system to guide him in acquiring new knowledge and new skill. These systems can be further divided according to their methods of reasoning about the user into those using active analysis, such as GREATERP, GIL and DISCOVER, and those using passive analysis, such as Bridge.

Interactive approach and immediacy features are two main characteristics which differentiate active systems from passive ones. Through rich interactions and immediate feedback, active systems bring affordance into users' knowledge of perception and action. Users are expected to perceive and manipulate the dynamic behavior of the program and its unfolding computation with less effort and accelerate the debugging task. Hence, these systems can reduce the mental overhead and help the users in putting more emphasis on program understanding, debugging and problem-solving.

The remainder of this section examines in more detail these four fully implemented systems that perform in-event analysis. These examples are intended to show that every one of these automatic program debugging and tutoring systems is limited due to its inability either to support sufficiently large grain size of automatic debugging, as in the case of GREATERP, or to support active debugging of partial solution steps as they are provided by the user, as in the case of Bridge. A more critical analysis of the Lisp Tutor vs. DISCOVER is covered in the last section of this paper.

3.2.1 Greaterp

GREATERP (the Lisp Tutor) uses production rules to synthesize code for both an 'ideal' and a 'buggy' novice Lisp programmer. The student's design decisions are traced as the student develops the program. This approach is called the model tracing approach to automatic program debugging. As each Lisp symbol in the student program is entered, GREATERP decides what rule would have to fire to duplicate the input. If the duplicating rule is in the 'expert' set then GREATERP does nothing, but if the duplicating rule is in the 'buggy' set then GREATERP gives the student a short tutorial on his misconception. In this way, the system always checks to see if the student is following a design path of an ideal model. Buggy paths are pruned as soon as they are discovered.

The system is based on a cognitive model which suggests that mistakes by the student should be flagged as soon as they are encountered. Anderson (1990) argues that there is considerable psychological evidence that humans learn better with immediate feedback (an important learning assumption of the ACT* theory of skill acquisition), and that this approach increases the possibility that the student will be able to relate the advice to the current issue (context) rather than getting advice at a much later stage.

The system is very directive and interventionist. Computational advantages of such an approach are that it reduces the usual combinatorics associated with mal-rules to decide whether or not each new Lisp

token is a legal or illegal continuation of the program, and that it eliminates the problem of multiple bugs. The problem of deciding which bug to tutor does not even arise, because the tutor simply prevents the student from making multiple bugs. Though of course, it cannot prevent the user from entertaining multiple misconceptions about a single symbol. The disadvantage of this approach is that the student is highly constrained in the solutions that can be developed. The student must conform to the task decomposition and coding sequence that GREATERP enforces.

The latest version of GREATERP attempts to overcome some of the problems cited above by varying the nature of the tutorial interaction. The new system has employed a problem compilation approach to provide the student with more control over the coding process in two ways: by relaxing the constraint on , input order, so that the student can generate code in any order he wants instead of the left-to-right, top-to-bottom manner, and by giving the student control over when feedback is presented. This student controlled feedback is achieved by delaying feeding each unit of code to the tutorial engine as it is generated. Instead, the code is buffered and submitted to the tutor at the student's request. However, the debugging of the user program is carried out in the same way: the system scans the code in a top-to-bottom, left-to-right manner, stops at the first error encountered, and ignores the rest of the solution.

This transition to the student-controlled interaction makes the new system, like Bridge, a passive system that waits for the novice to request automatic analysis of his code, and thus loses the rich interaction with him. By doing that, this new system loses very important features: the ability to monitor the novice's progress on the solution path, determine when he shows evidence of errors within their proper and immediate context and decide when to guide him in what to say during interactive tutoring. In principle, there is no reason why a model-tracing system should stick to a single-symbol based immediate feedback. One possible alternative would be to increase the grain size of automatic debugging to a full program statement or even to a block of statements, not just a single Lisp symbol. This would not only provide the system with an ability to delay its interactive feedback but would also allow the user to backtrack and delete some previously entered code and restart. Thus the system would support a more flexible style of interaction while preserving strong ties to the cognitive principles of model-tracing and immediate feedback.

When two design paths leading to alternate implementations overlapped, GREATERP, in its old version, could not determine which design path was being followed. Now through problem compilation approach, GREATERP can search multiple alternative branches down in the goal tree when it needs to disambiguate some responses. In the current version, discrepancies between predicted implementations and expected implementations can be explained either in terms of an incorrect design path being assumed, or in terms of a bug in the student's program. By doing that, GREATERP follows a similar approach to interpreting discrepancies as that of Proust: either the wrong plan has been chosen to interpret the program, or the plan is correct but the student's program has a bug.

The student interacts with the system mainly through a structure editor. This feature aids automatic debugging by eliminating certain low-level syntax errors, such as balancing parentheses, and trapping and immediately remedying others, such as quoting a function call. This feature also ensures that both the system and the student know explicitly which part of the problem is being coded, since this is always done by replacing one of the place-holding parameters in the partially completed code. In this way, the student knows most of the time what goals and subgoals need to be satisfied.

Besides the code-level interaction, the system also supports a planning mode. This mode is used by the system when the knowledge of the student's design and planning decisions becomes very difficult to be derived from the code entered by the student. The mode is also used when the system suspects that the student is having some difficulty in planning or when the student requests help. Interactions at the planning-level are supported by multiple choices from the menus. This feature simplifies the task of automatic debugging even more, since in this way plans are more easily recognized than trying to derive them from step-by-step coding.

3.2.2 Gil

It is worth noting that the other model-tracing system, GIL, does not defer judgment. Any errors are immediately pointed out and the feedback is presented to the user as soon as a single erroneous token is encountered. However, GIL has one clear advantage over GREATERP. The system incorporates a visual programming environment which allows the user to observe in a graphical manner how Lisp lists are constructed. The newer version of GIL extends its functionality even more. This is achieved by augmenting the guided, visible environment with a free, visible environment. By doing that, this newer version has clearly followed a path similar to that already taken by the DISCOVER system.

Like DISCOVER, this recent version of GIL synthesizes a free programming environment with a guided programming environment. This is implemented in two phases. During the first phase (the free programming phase), the users are encouraged to explore the graphical programming environment to build a mental model of simple Lisp functions. Users build a program by connecting together objects that represent program constructs into a graph, rather than by defining Lisp functions in their traditional text form. However, the environment is not as dynamic as that of DISCOVER's. GIL does not evaluate individual steps taken by the users during their exploration, and hence does not allow the users to see the immediate effects of their actions in relation to the behavior of Lisp functions. For example, when the user selects the 'CONS' icon from the menu, GIL displays a box with 'CONS' written in it along with two branches coming out from the box. Here the user is expected to provide two arguments that go with the selected function. After filling the two branches with appropriate arguments, the user does not get to see how the selected function is applied to the arguments. The system just displays the partially completed graph. It would be nice if GIL could allow the users to see the dynamic behavior of the partially completed graph.

During the second phase (the guided phase), GIL's users solve simple programming problems under the intelligent guidance of the system. To be able to trace the user's solution, the system requires the user to specify his next step by selecting graphical icons from the menu that correspond to Lisp functions. Concerning model-tracing, GIL analyzes each and every single Lisp symbol provided by the user. However, GIL has one important advantage over both DISCOVER and the Lisp Tutor: it allows more flexibility than working on a program in top-down, left-to-right order. This is accomplished by supporting both forward and backward reasoning, and thus allowing the user to choose the part of the problem on which to work.

3.2.3 Bridge

Bridge is a programming environment intended to understand student design and partially complete programs. The system provides the user with intermediate design languages that allow him to talk about his plans and intentions directly. By doing that, the system avoids the process of deriving student intentions using complicated techniques such as partial matching based on a bug catalog used in the Proust system, or partial matching based on a reference solution used in DISCOVER, or a process model of the student's decision making used in GREATERP and GIL. Bridge supports the user in an initial informal statement of a problem solution, later refinement of that solution, and final implementation of the solution as programming language code. This is done in three phases. In phase one, the user constructs a set of step-by-step instructions for other people. Each phrase represents a goal and corresponds to one programming plan. In the next phase, the user matches these phrases to programming plans and builds a program using a representation of these plans. In phase three, the user matches these plans to actual programming language constructs of Pascal and builds a solution to the original problem.

As an example of this goal-plan-code matching during theses three phases, consider the following simple problem. *Write a program to get an integer from the user and output the result of multiplying the integer by 10.* To solve this problem correctly, the user is expected to provide the following four goals which represent one way to solve the problem:

Setup the memory variables for the integer and its result *(G1)*

Ask the user to enter an integer *(G2)*
Multiply the integer by 10 *(G3)*
Output the result on the display *(G4)*

Once the goals are provided, the user can now select the plans which would satisfy the above goals, i.e. plans that would implement the goals. Here is one possible scenario:

Declare Num1 and Resultnum *(P1)*
Input Num1 *(P2)*
*Resultnum = Num1 * 10* *(P3)*
Output Resultnum *(P4)*

Finally, the user matches these plans to actual programming language constructs of Pascal and builds a solution to the original problem. After the user finishes building a natural language solution to the problem, the system builds a symbolic representation of the natural language version of the program. In this representation, Bridge notes the order of each plan, and compares the representation with a list of requirements for a correct solution to the problem. The first requirement that the student fails to satisfy becomes the subject of the tutor's remarks. In other words, Bridge compares the required plans and their ordering in a pre-stored list of requirements against the plans supplied by the user in his solution. Whenever a mismatch occurs in this comparison, the system presents the user with a hind. Only when all the plans in the requirements list are satisfied is the user allowed to move to the next phase. The system also supports some aspects of supportive environments (microworlds), especially in the second phase. This is accomplished by highlighting each plan during execution, animating data flow with floating value plans and highlighting the corresponding program statements, written during the first phase using the intermediate design language of the system.

The advantages of providing a language for intermediate representations to allow the student to express his planning actions are obvious: (1) it provides the student with specific mental models with which to conceptualize the problem-solving process, and (2) it separates out the two activities of planning and coding, and thus reduces the cognitive load and allows focusing on one level of problem-solving at a time. This feature also reduces the engineering cost of automatic debugging, since the system does not need to infer the plans from the code using complex pattern matching: the student spells them out to the system through menu selections.

The driving force behind the design of the Bridge system was the desire of its developers to come up with an interactive learning programming environment that would overcome (1) the lack of rich interaction found in the Proust system, and (2) the rigidity and restricting directiveness found in GREATERP. The current implementation of the system, however, is not as interactive as it is liked to be seen. The system has no capability of actively reasoning about the novice user while he develops the solution. During the first phase, novices are expected to select a plan that correctly goes with a goal statement and to relate these plans together in a correct order. This pre-assumes that the user possesses adequate algorithmic skill to be able to accomplish this task. When the user requests automatic debugging of his solution, the system steps through the code and presents the user with its feedback. By doing that, Bridge resembles the latest version of GREATERP (the student-controlled version). An interesting extension to the system would be to make it capable of explicitly guiding the novice in the process of selecting programming plans to go with the goals and organizing these plans in a required order.

The important assumption underlying the system is that teaching plans to novices will improve their ability to comprehend and write programs. There is very little evidence to suggest that this method is the only way that can provide a basis for the kind of outcome that Bonar (1992) hopes. Experiments looking at using plans across different tasks and programming languages have demonstrated that other knowledge structures play an equal or even greater role in programming comprehension and generation than plans (Gilmore *et al*, 1988). The evaluation of Bridge focuses on informal reports of the problems encountered

during interactions with the system and, therefore, provides neither a good insight into its effectiveness for novice programming nor an account on the usefulness of the plan-based approach to the design of learning environments.

In the second phase, novices are required to select a plan and then select a corresponding Pascal construct that best implements that plan. One of the difficulties of this approach is that the mapping from a plan to a piece of code, such as an assignment, or vice versa is neither easy for nor clear to the novices. Here is an assumption that the novice user already knows enough about these plans to be able to successfully select from them the most appropriate one for the next mapping task. Interestingly, Bonar (1992) reports that the second phase was the most problematic in that the novices found it quite difficult to translate their plan-based solutions into Pascal programs. This appears to suggest that knowing plans only might not be adequate to explain the performance of novice programmers. Rather, what appears to be important is the conceptual knowledge and understanding of how different programming plans are used. One possible way to simplify this task would be to provide novices with a microworld-like, plan-based environment through which they can 'discover' themselves how different plans are used.

3.2.4 Discover

Like GIL, DISCOVER is an intelligent programming environment which synthesizes free with guided programming and supports software visualization (Mukherjea *et al*, 1994; Stasko *et al*, 1992; Price, 1993; Baecker *et al*, 1997; Anjaneyulu, 1992) and immediacy of feedback. The system is designed to help novices acquire both programming knowledge and programming skill. This is accomplished in two phases:

- In the first phase, the exploratory phase, the system helps novices through visualization and immediacy features to explore the dynamic behavior of programming statements and of the underlying notional machine to build a robust mental model of language execution and machine behavior.
- In the second phase, the guided phase, novices put together program statements and language constructs, explored in the first phase, to solve problems under the intelligent guidance of DISCOVER.

The interface of the DISCOVER system (shown in figure 1) was designed to facilitate the comprehension of large quantities and activity options that normally characterize discovery systems. The interface appears to a user as a collection of seven windows. The four windows on the left side of the interface, namely the *Memory Space, the Input Space, the Output Space* and the *Algorithm Space,* represent the components of the programming machine (Ramadhan, 2000b).

The interface is designed to expose the users to whatever is being manipulated and experienced, and hence brings them closer to the language and the machine. By doing that, the system brings the affordance into the users' knowledge of perception and action. The visual model of the notional machine, through visualisation and immediacy features, is expected to help the users perceive and manipulate the dynamic behavior of the program and its unfolding computation with less effort and accelerate the debugging task. By allowing the users to have such visual view of program behavior within an integrated and coherent image of the programming machine, the system can reduce the mental overload on the novice users, hence more emphasis can be put on program understanding, debugging and problem solving. In short, interactions in DISCOVER among the program, the language and the machine are designed to produce an environment which makes it effortless for users to examine a program, figure out its effects and connections, and to relate problem solving with the properties of the machine they are interacting with.

It may be argued that the interface with its multiple windows and various forms of feedback messages may increase the cognitive overload on the novice users during learning programming. We are aware of this point. During our pilot evaluations, summarized in the last section of this paper, we paid special attention to the comments made by the users on the usefulness of the display and its organization. We did not find any serious concerns on the novices' side regarding the interface. We also have no reason to suspect that the interface layout is somewhat confusing to novices or distracting them while using DISCOVER. In fact, it is worth noting that the entire left portion of the interface represents one logical

entity which nicely integrates its four components and provides a high-level image of the underlying programming machine.

The programming language of the system is a simple pseudo-code based and algorithm-like language. At present, the language has no provision for functions, procedures, recursion and complicated data structures such as records, arrays and lists, thus focusing users' attention on basic programming concepts and simplifying the learning process. Programming concepts supported include CREATE, PUT, READ IN, WRITE OUT, WHILE-END-WHILE and IF-ISTRUE-ISFALSE. The naming of these concepts was drawn from the results of several well organized empirical experiments which studied the effects of pseudo-code as a programming language (Vessey, 1985; Boehm-Davis, 1987; Curtis, 198; Dyck, 1987; Mayer, 1985).

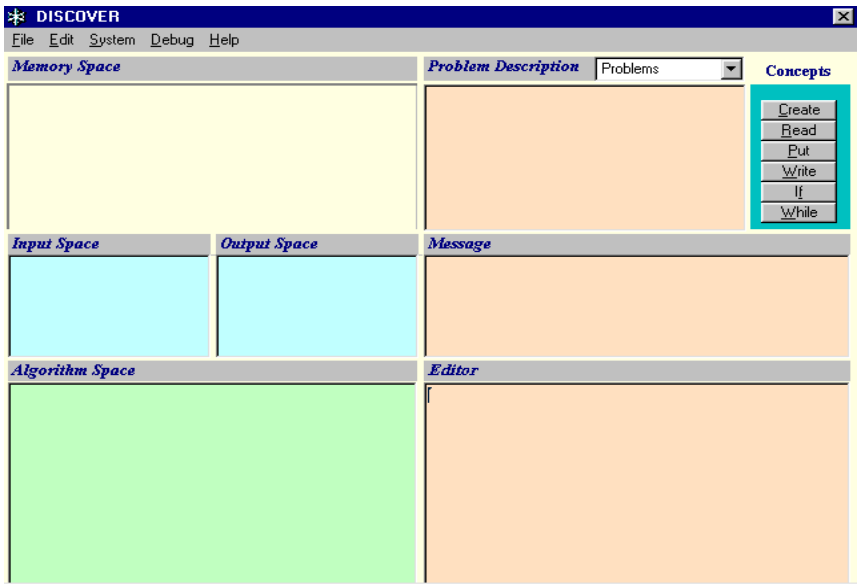

Figure 1: The interface

It may be claimed that the language is very simple and that more functionality needs to be supported by the language to orientate novices to modern programming environments. We accept this argument and note that there is no reason why DISCOVER's language should not be scaled up to support advanced users. This can be done by supporting advanced programming features such as procedures, functions and recursion, and more abstract data structures such as arrays, records, pointers, sets and files. In fact, it would be interesting to see in future versions of DISCOVER visual representations of advanced data structures such as lists and trees in the *Memory Space* of the system. Having said that, it is our firm conviction that environments designed for beginner and intermediate programmers should avoid the temptation of including complicated functions and structures found in modern commercial languages such as support for visual applications, device interfacing, parallel programming and network support. Clearly, with this advanced functionality, the environment would no longer be suitable for novice programmers.

The selections in the *Concepts* menu, shown in the top right-most position of the screen, contain the beginning of phrases. Each phrase corresponds to one programming concept. When the user selects a concept, its corresponding name is inserted into the *Editor* window and all the user has to do is to complete it by typing in its parameter part (e.g. the names of memory cells to be created). A template showing the

correct syntax for the selected concept is also inserted in the window next to the *Concepts* menu. Once a concept is completed it appears in the *Algorithm Space* window, where the code so far entered is stored.

During the free phase, users can either load pre-written example programs or type in their own programs and visually observe the dynamic behavior of the language and the machine. For the later option, the system interprets each and every statement immediately after it is completed and visually shows its effects on the components of the notional machine. The user can also disable this interactive execution of his statements. In this case, the user can type his program and request the execution at any point by selecting the *Run* option from the *Debug* menu. Syntax errors in the programs are trapped by the syntax-directed editor and reported in the *Message* window. Figure 2 shows the visual execution of a program loaded by the user.

The aim of the guided phase is to teach the novice how to compose and co-ordinate programming concepts and statements to solve programming problems under the intelligent guidance of the system and thus build effective problem-solving skills. The user has to build his solution to the current problem by properly putting together programming concepts. Once a concept is completed and accepted by the syntax-directed editor it is passed to the intelligent component of DISCOVER for automatic diagnosis. In doing so, DISCOVER attempts to model the steps taken by the user by evaluating his actions and responses. DISCOVER analyzes the surface code of the completed statement (partial solution code) without much specific knowledge about the problem to be solved or about how to design and construct an algorithm (i.e. DISCOVER cannot solve the problem itself). During the guided phase, users select from the *Problems* menu the problem to be solved. For each problem, a statement is presented in the *Problem Description* window.

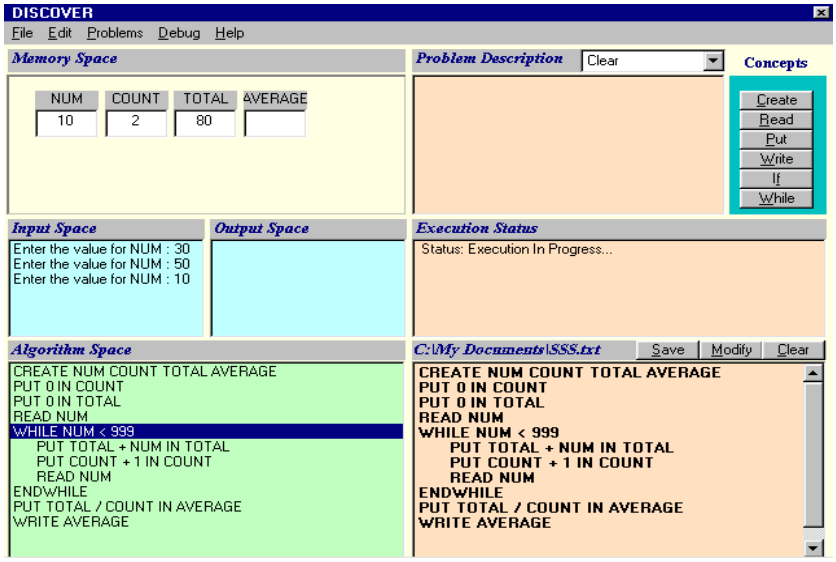

Figure 2: Visual execution of a program

In both phases, the system visually executes the currently encountered statement and instantly shows the changes that take place in the *Memory Space, Input Space or Output Space* windows. In addition, each statement of the program is visually highlighted during the execution to show the user how the control flows from one statement to another, and how the highlighted statement affects the current state of the underlying notional machine. Figure 3 shows an example of interacting with the novice user during the

guided phase. This figure also shows how different programming concepts affect the components of the visible notional machine during execution in a manner which clearly shows possible causes and effects.

This figure shows the user interaction while attempting to solve the *Ending Value Averaging Problem.* In this example, the user has failed to accumulate the numbers read by the program for the averaging purpose. The system detects this misconception and considers this step as a deviation from the solution path, and hence decides to interfere by guiding the user toward the expected step. For more detail regarding the diagnosis process, see (Ramadhan, 1997).

Much like Bridge, DISCOVER relies on a pre-stored reference solution (the ideal student model) for a given problem and applies various heuristics and pattern matching techniques to match the solution code provided by the novice with the reference solution in order to spot errors and misconceptions. However, like GREATERP and GIL, DISCOVER is a model-tracing based system, capable of interactively analyzing partial solution code and providing immediate feedback on failure. The system explicitly guides the novice in the process of putting together programming concepts to solve the given problem. DISCOVER monitors the novices actions, not on a symbol-by-symbol basis as it is done in GREATERP, but on a complete statement-by-statement basis. As long as each statement represents a correct goal on a solution path, DISCOVER continues guiding the novice towards the final goal, reasoning about the goals already satisfied and hinting at the goals that still remain to be satisfied.

Unlike GREATERP and GIL, however, DISCOVER (1) utilizes goals and plans (not a production system) to represent the domain expertise, (2) does not keep an account, at least currently, of common error patterns (the buggy model), and (3) supports an ability to give delayed feedback by increasing the grain size of automatic debugging to a complete programming statement (not just a single symbol or token) and an ability to do limited backtracking by giving the user some chance to delete previously entered code and restart. See (Ramadhan, 1997) for more description on how DISCOVER represents the reference solution and how it implements the diagnosis process. In the sections to follow, we present a more detailed discussion on various design issues incorporated in active program diagnosis systems along with suggestions for improving their engineering. As mentioned earlier, this discussion is presented in the context of the system DISCOVER.

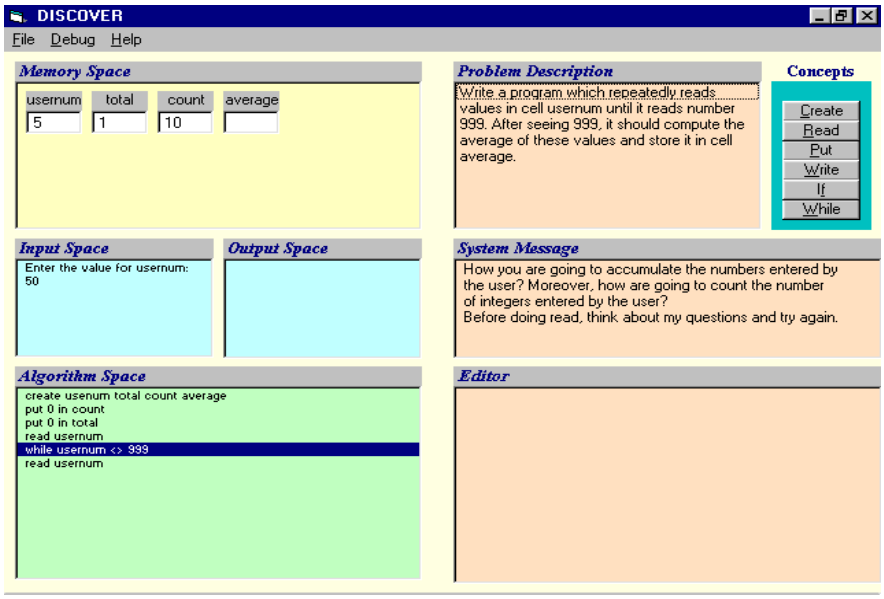

Figure 3 : A message from the intelligent component

3.2.5 A Summarized Comparative Analysis

This section provides a short analysis of the four systems included in the *in-event analysis* category. An objective and comparative evaluation of learning environments requires a set of standards to measure the effectiveness of the systems. To compare the features of these systems reviewed above, we use the following three design principles as criteria for our analysis:

1. Systems should help the users in avoiding learning a mass of detailed syntax of the programming language being explored.
2. Systems should assist the users in exploring the programming experience interactively through supporting various immediacy features. Users should become active learners: forming their own hypotheses, exploring their own questions, and drawing their own conclusions.
3. Systems should assist the users in problem-solving through supporting automatic diagnosis with flexible immediate feedback on individual errors.

Table 1 shows the correspondence between the criteria established above and the four systems surveyed in the previous subsections.

Table 1: Comparative analysis

Systems	Criteria 1	Criteria 2	Criteria 3
isp Tutor	Fair	Excellent	Good
Bridge	Excellent	Good	Fair
GIL	Fair	Excellent	Good
D SCOVER	Excellent	Excellent	Good

Regarding the first criterion, namely the scope of the language being explored, It has been claimed that simplicity is among the most important characteristics that a programming language intended for novices should have (du Boulay *et al*, 1981). Three types of simplicity have been proposed: functional, logical and syntactic simplicity.

The language should be kept functionally simple by giving it small set of basic instructions that are easy to understand; it should be kept logically simple by giving it instructions that are suited to the problems of interest to the novices, so that they can tackle and solve these problems by short and simple programs; and it should be kept syntactically simple, i.e. the rules for writing instructions should be uniform and have well chosen names. The names of the basic instructions are important as novices tend to make inferences about the language from these names. Examples of these names are LOAD and STORE instructions used in the assembler language that have real world connotations. Both the Lisp Tutor and GIL require the users to have a good command of the Lisp language. Because of high abstraction involved in the syntax and semantics of the Lisp commands, constructs, and functions, users are faced with greater mental overhead when learning to program using the Lisp language.

DISCOVER and Bridge use a pseudo-code like language which puts more emphasis on programming concepts and less emphasis on detailed syntax and abstract data types and constructs. Regarding the pseudo-code language, it has been noted that comprehension of programs depend both on the language used and on the task for which the language is used (du Boulay *et al*, 1981). Thus, even if the underlying algorithms are identical, creating the internal representation from one language may be more difficult than creating that representation from another language. In addition, it has been reported that the reason why it may be less difficult to create internal representations using pseudo-code language could be that it lessens the 'translation distance' from the documentation format to the program code.

On the second Criterion, namely the scope of the interaction, DISCOVER, GIL and the Lisp Tutor provide highly interactive environments. All these three systems implement program diagnosis using model-tracing methodology, by which they monitor the actions of the user as he moves along the solution

path, automatically analyze partial solutions for semantic errors and misconceptions, and offer guidance whenever he deviates from a correct solution path. Bridge is not as interactive as it should be. The system has no capability of actively reasoning about the novice user while he develops the solution. As it was mentioned in section 3.2.3, the user builds his solution in a passive-like mode. When the user requests automatic debugging of his solution, the system steps through the code and presents the user with its feedback. An interesting extension to the system would be to make it capable of explicitly guiding the novice in the process of selecting programming plans to go with the goals and organizing these plans in a required order.

Finally, on the third criterion, namely flexibility of the immediate feedback, Bridge falls behind the other three systems, as shown in table 1. The system has no capability of actively reasoning about the novice user while developing the solution, and hence the feedback is not immediate. DISCOVER, GIL and the Lisp Tutor do support immediate feedback on errors. However, the grain size of automatic diagnosis in DISCOVER is not confined to a single language token or command, as in GIL and the Lisp Tutor, but to a complete statement and expression. This feature gives the user some opportunity for self-correction and provides a larger context for tutorial instruction; hence its feedback is considered to be more flexible.

4. Model-Tracing Based Diagnosis

Model-tracing, used in the Lisp Tutor and GIL, simply expresses the fact that the novice user is made to follow the system's model quite closely. A model-tracing based system analyzes each and every step of the user's solution to determine whether it is on a correct path toward a solution or indicates a misconception. In the Lisp Tutor and GIL, the user's step is analyzed by comparing it with the rules currently considered by the system, which represent the ideal user model.

If the step taken by the user is one that can be produced by executing one of the rules in the ideal user model, the rule is applied and the user is considered to be moving on a correct solution path. In this case, the system remains silent in the background and permits the user to continue. Alternatively, if the user's step cannot be produced by the ideal model, the system considers its buggy model, which represents general patterns of errors. Misconceptions are flagged and diagnosed when the user's step is produced by one of the rules of the buggy model. Here the system interrupts and offers advice associated with the buggy rule. In this way, the system understands each step the user takes to build his program. It is this combined use of the ideal and buggy models, together called the generic model, which defines the model-tracing methodology: the system traces out the path currently taken by the user through the generic model and insists that the user stay on a correct path.

In short, the main features of model-tracing based diagnosis and tutoring are the following:

- The system constantly monitors partial steps taken by the user and intervenes whenever he shows an evidence for a misconception by deviating from a solution path.
- The interface in these systems tries to eliminate the problem of checking low-level syntax of the language being learned (e.g. via the use of structure editors), and thus reduces the mental overhead associated with problem-solving.
- The interface is highly active in that it responds to every step (e.g. a single Lisp symbol) the user provides.

Through this approach to automatic diagnosis and tutoring, a model-tracing system can (1) diagnose very specific errors and misconceptions, and provide clear advice and explanation within proper and immediate context, (2) explicitly guide the user in the process of organizing different programming concepts and statements, and (3) simplify the engineering of automatic diagnosis by preventing multiple bugs and errors.

Anderson's (1990) work on model-tracing and immediate feedback has strongly argued that these advantages make it worthwhile to incorporate this approach in the design of intelligent diagnosis and tutoring systems. His well documented empirical evaluations of the effectiveness of model-tracing and immediate feedback in procedural domain indicate that users learn procedures more quickly than conventional tutoring when provided with a model-tracing based environment. Students can more easily utilize feedback and explanations when the system, the Lisp Tutor, supports the capability of automatically tracing, analyzing and reasoning about their partial solution steps that led to the error. Anderson has also shown that such a tutoring strategy can prevent long episodes of counter-productive floundering by interactively trapping errors and correcting them as they show up in their proper context during the performance of a task. Similar results have also been reported by Reiser (1992) on his model-tracing, interactive, graphical Lisp tutoring system GIL.

4.1 Critiquing Model-Tracing

Despite the achievement of the model-tracing based methodology to intelligent diagnosis, the approach suffers several drawbacks and shortcomings. First, by restricting the user to a symbol-by-symbol based top-to-bottom coding order, model-tracing hinders the opportunity for experimentation that might lead to a clearer understanding of the problem and thereby does not allow the users to explore and discover new strategies nor does it allow them to detect and correct their own errors and misconceptions. The main driving force behind model-tracing based systems is the detection of deviations from the ideal user model. These systems reject any other correct approach to solving a problem if it differs from the path currently followed by the system (Nwana, 1991; Wegner, 1987).

Second, the success of model-tracing based systems depends heavily on the extent of their model-tracing knowledge, which includes the number of correct rules in the ideal user model and the number of the mal-rules in the buggy user model (Wegner, 1987). For example, the production system of the Lisp Tutor currently contains more than 1200 rules, more than half of which are mal-rules (Anderson, 1990). Production systems, despite their many advantages, impose several computational problems when utilized to support model-tracing based diagnosis and tutoring, especially of large problems, see section 4.3.2.

Third, the important programming activity of debugging is taken away from the user since model-tracing based systems, in principle, do not permit floundering (Nwana, 1991; Wegner, 1987). As a consequence, such systems may weaken the user's personal motivation and sense of discovery. To address these issues, we have designed a prototype system called DISCOVER. The system uses a different implementation of model-tracing, supports an improved engineering of model-tracing based diagnosis, and provides a slightly more flexible style of tutorial interaction (e.g. than the Lisp Tutor) while preserving close ties to the underlying cognitive modeling of the model-tracing based diagnosis. Before moving on, it is very important to clarify points that have to do with the terminology used to describe DISCOVER's approach to automatic diagnosis, intelligent tutoring and knowledge representation.

4.1.1 What Goals and Plans in DISCOVER Stand for

There has been a great deal of work with programming 'plans'. This includes formal definition (Rich, 1986), empirical investigation (Soloway, 1984), and implementation in AI systems (Johnson, 1990). The plan notation has been developed to characterize aspects of language independent knowledge (information) about programming and to characterize language dependent knowledge used by novices and experts to implement programming problems (Bonar, 1992).

In this paper, we focus on plans as a tool for coding and matching syntactic templates that correspond to the low-level syntactic code objects in the DISCOVER's language. This implies that plans here are language dependent. Moreover, the plans presented and used in this paper reflect only the actual implementation of the code provided by novices, and *not* the possible implementation of the underlying knowledge of novices. The plans used in this research are thus neither (1) characterized to be the plans that were derived from a psychological theory of programming plans that was developed at Yale, or the plans

that were developed at MIT for the automatic programming project (Rich, 1986), nor (2) characterized to have any sort of structures in them.

Concerning the 'goals,' Johnson (1990) points out that goals are the principal requirements that must be satisfied if a solution is to meet the problem specification. These goals are used to represent the intentions of the programmer. A goal statement consists of a name of a type of goal followed by arguments. In short, Johnson argues that goals describe what the programs must do, while plans describe how these programs are supposed to do it. In this paper, the term 'goal' is used to indicate the programming concept that the user is expected to select while solving a programming problem. So for example, when we say that the next goal is to *initialize cell NUM*, what we mean is that the user is expected to select a PUT concept from the menu to store a value in cell NUM. In this paper, goals are used as a planning tool for enabling novices to break down the problem in smaller steps, and each corresponds to the concept in a single program statement. In short, goals are mapped to programming concepts, which in turn are mapped to single statements. Whereas in the original goal-and-plan literature, goals are mapped to plans, which are mapped to structures, which in turn are mapped to multiple statements, or possibly single statements.

4.1.2 Is DISCOVER an Intelligent Diagnosis and Tutoring System?

Instead of using a production system to perform automatic diagnosis and tutoring, DISCOVER uses a reference solution to trace all the possible solution paths needed to guide the tutoring process. The reference solution is represented in terms of a Proust-like goal-and-plan tree, except (1) that it includes explicit relationships to constrain user steps on a solution path, and thus allows DISCOVER to preserve strong ties with the model-tracing paradigm, and (2) it is encoded procedurally. In the current implementation, the reference solutions (goal-and-plan trees) that guide automatic diagnosis in DISCOVER are generated by hand, and manually coded as a set of procedures, where a procedure represents a goal or subgoal, which in turn calls other procedures that represent the plans. In short, at present there is no explicit representation of knowledge in the system.

It could be argued that this approach neither makes DISCOVER a knowledge-based system nor an intelligent debugging system, and hence a better term to characterize DISCOVER would be 'an automatic diagnosis system'. While we agree with the first criticism, we argue that DISCOVER is more than an automatic diagnosis system. Since the system currently has no student model and good courseware, we do not claim that it is a complete ITS. The plans in the reference tree have provision for associating a feedback message with user behavior. The system through these hand-coded explanations is capable of generating highly specific tutorial-like messages, and thus simulating some of the functionality of an ITS. By supporting that, DISCOVER is more than just a diagnosis system.

4.1.3 Production Systems and Model-Tracing

In this paper, we argue that a production system is not the *most* efficient way to represent the domain expertise needed to implement model-tracing. Model-tracing requires the system to be capable of interactively tracing all the possible paths that a user might decide to follow. To accomplish this, a production system needs to follow several branches simultaneously, and thus keep several rules active at one time. This is true even when a model-tracing system, such as the Lisp Tutor, is only capable of handling a very small grain size of modeling (e.g. single Lisp symbol). In this paper, we argue that for a model-tracing system to be able to handle larger grain sizes of diagnosis (e.g. a complete program statement), the computational cost associated with time and space becomes too high. It is important to note the context for the discussion that follows. The critique is of production rules as used in model-tracing systems like the Lisp Tutor. It is not a critique of production systems in general. In short, the discussion that follows does not attempt to devalue the modular and expressive nature of production systems, but the way these systems are used to handle model-tracing in the domain of computer programming.

4.2 DISCOVER Approach to Model-Tracing

Like the Lisp Tutor and GIL, DISCOVER analyzes each and every step of the user's solution to determine whether it is on a correct path toward a solution or indicates a misconception. However, the grain size of automatic diagnosis in DISCOVER is not confined to a single language token or command, but to a complete statement and expression. This feature gives the user some opportunity for self-correction and provides a larger context for tutorial instruction.

To consider a simple example, suppose the user is expected to compute the average by generating 'PUT total/count IN average' statement in DISCOVER language. DISCOVER will not diagnose individual parameters and tokens that make up this expression and will wait until the user submits the completed statement as his current step by hitting the return key. Even if the user selects an entirely different concept than the one expected by the system, for example 'READ' instead of 'PUT' in this case, the system will immediately recognize the bad selection but will not flag an error and thus give the user some opportunity for self-correction.

In DISCOVER, the user's step is analyzed by comparing it with the goals and plans of the reference solution, and not rules. If the step taken by the user is the one that can be matched with one of the plans (e.g. with the syntactic templates of plans) that are currently considered by the system, the plan is applied (e.g. the plan's template is matched with the code object in the user's statement) and the user is considered to be moving on a correct solution path. In this case, the system permits him to continue. Alternatively, if the user's step cannot be matched with any of the plans currently considered by the system, the system interrupts and offers a feedback message that attempts to explain the misconception in relation to its current context.

In short, DISCOVER relies on its explicit planning mechanism (discussed in the next section) to trace the user's planning and design decisions during problem-solving. As each complete statement in the user's program is entered, DISCOVER checks to see if the user is following a correct design path. Incorrect paths are pruned as soon as they are detected and the user is allowed to try again. If the user cannot determine how to proceed, DISCOVER can assist him and if necessary can provide the next correct step.

4.3 Improving the Engineering of Model-Tracing

Several alternative design principles can be proposed to tackle some of the pitfalls associated with the model-tracing approach, as it is implemented in the Lisp Tutor and GIL, while preserving close ties to the underlying cognitive modeling on which it is based. One approach would be to support the following features and capabilities:

1. Increasing the grain size of automatic analysis and tutoring to handle a complete expression and statement, rather than a single token, and thus delaying the feedback until the whole statement is submitted will give the user some flexibility for self-correction of errors. This approach will also provide the system a larger context for automatic diagnosis. This larger context in turn will enable the system to support a more flexible mode of tutorial interaction.
2. Supporting an explicit planning (though low level) mechanism through which the information about the user's planning and design actions are provided to the system naturally and voluntarily using a menu during all stages of the problem-solving process and not only in response to the Lisp Tutor like interventionist dialogue, which occurs at the last stage of the diagnosis process (e.g. when the system fails to determine the user's planning decisions) will not only provide the user with an opportunity to decompose the problem into smaller steps, but will also simplify the computational cost involved in the automatic diagnosis process.
3. Representing the ideal model using some other knowledge representation formalism that is more practical in terms of implementation and less expensive in terms of cost associated with time and space than the production system currently used in the Lisp Tutor and GIL.

4.3.1 Explicit Planning Mechanism

To model the problem-solving process, DISCOVER utilizes an explicit planning mechanism. Through this mechanism, DISCOVER, like Bridge and GIL, explicitly requires the novice user to select programming concepts from a menu. These selections externalize and represent the user's design actions. In this way, the information about the user's overall planning decisions is provided to the system by the user naturally while solving a problem. In other words, the information about the user's goals (e.g. programming concepts to be selected in this case) needed to monitor his progress on a solution path is provided nonintrusively as an integrated part of problem-solving.

The novice is presented with a menu of programming concepts that represent high-level goals. The novice solves the problem by selecting these concepts and putting them together in their proper positions. Selecting a READ IN concept, for example, indicates to the system that the novice's current goal is probably to get a value or an input from the user. Through this mechanism, the system always gets the information needed to trace the novice's actions in building the program.

This approach greatly simplifies the problems associated with the automatic diagnosis of the solution. The system does not need to establish goals (programming concepts) because the novice spells them out for it. Thus, the time spent by the system in diagnosing the errors could be certainly reduced, since the uncertainty in what path the novice would take is greatly minimized. The system compares the concept selected by the novice, which represents his current goal, with the one expected by the system (considered in the reference solution) and generates feedback without relying on a bug catalog.

This mechanism, however, does not eliminate the plan-recognition problem. The representation of DISCOVER's reference solution resembles the goal-and-plan tree of Spohrer (1985). For each goal, there is number of plans that may be applied to implement and satisfy that goal. Although the novice tells the system what goal he wants to pursue (i.e. what concept he wants to select), the system still needs to recognize the plans (i.e. code objects) used by the novice to properly implement the selected goal.

4.3.2 Plan-Based Model-Tracing

A frequently used strategy in representing domain knowledge is to use a set of problem solving rules. Each rule contains a description of a particular problem situation and a step to take in that situation, basically an action-oriented approach (Clancy, 1987). A combination of these rules makes up what is known to be the production system. A production based system traces a user's solution by matching each partial step provided by the user against the conditions of the rules in its problem solving model. GIL and Lisp Tutor are the classic examples of programming tutoring systems that follow this approach.

Production systems, despite their advantages, are not the most efficient way (e.g. in terms of computational costs associated with time and space (Anderson, 1990) to implement model-tracing (discussed in the next section). These systems have to consider a very large number of rules at any point during diagnosis process to be able to trace all possible next steps that the novice might follow. Moreover, to cope with the problem of nondeterminism, these systems have to be used nondeterministically (i.e. more than one rule active at once) to be able to trace multiple paths before disambiguating information is encountered.

The inability of these production systems to easily handle a larger grain size of modeling, for example a complete programming expression or statement (see next section), while supporting model-tracing, greatly contributes to their weaknesses. Theoretically, there is no reason why a production system cannot handle larger grain sizes of modeling. In practice though, this would require a large increase in the number of rules, as will be shown shortly. In fact, it is for this reason that the systems based on such representation force a particular interpretation of the novice's behavior on the novice (e.g. single-symbol based tutoring), rather than waiting until the novice generates enough of the solution step (e.g. complete statement), which in turn will enable the system to establish an adequate context for dealing with ambiguity. Therefore, to increase the grain size of tutoring, a model-tracing system needs to depart somehow from using production systems as its driving force during the process of automatic diagnosis and tutoring.

Implementing a production system also has high computational cost both in terms of space and time. Problems tend to become more costly as they become larger even if they involve the same underlying knowledge. This is because the working memory of the production system tends to increase, as does the nondeterminism. In terms of time, a production based tutoring system becomes very slow when trying to simulate the user dynamically and interactively in order to trace and guide him.

Running the production system through an off-line compiler would solve the computational cost associated with time, but would increase the computational cost associated with space. Because when the system is run ahead of time to produce the trace tree, it is necessary to follow every branch at an or-node so that later the system can trace the user down any possible branch. This requires exhaustively searching the trace tree for possible alternatives in the user solution, and also results in very large structures needed to store the trace tree. Additionally, it is also an onerous task to develop complete production systems that also include a good set of buggy rules to model possible misconceptions and errors (Nwana, 1991).

DISCOVER uses a different approach to implementing a model-tracing based tutoring. Instead of developing a complete production system with all the necessary mal-rules in it, DISCOVER uses a reference solution to trace all the possible solution paths needed to guide the diagnosis process. Currently, the system has no knowledge of what bugs and misconceptions are likely to occur in the novice program. The system relies on its explicit planning mechanism to trace the user's high-level goals and utilizes pattern matching and heuristics to trace the user's plan-oriented actions. Through this approach, DISCOVER detects and diagnoses very specific bugs when they arise in their immediate and proper context.

The reference solution is represented in terms of a Proust-like goal-and-plan tree, except that it includes explicit relationships to constrain user steps on a solution path, and thus allows DISCOVER to preserve strong ties with the model-tracing paradigm. In addition, plans in the reference tree also have provision for associating a feedback message with user behavior. Goals represent different programming concepts which the novice needs to have in his solution and the plans represent the correct implementation of goals. Thus plans are used to indicate the textual structure that the user code must have and the goal-subgoal structure of the code. Variability as well as constraints over the user solution are represented using AND/OR clauses in the reference tree. It is this representation coupled with pattern matching which makes DISCOVER capable of supporting more variability (e.g. than the Lisp Tutor and GIL) and larger grain sizes of modeling. For a detailed description of the representation mentioned here, see (Ramadhan, 1997).

4.3.3 Rules Versus Plans

It is not easy for a rule-based system to allow the type of variability supported by DISCOVER, while at the same time handling active diagnosis and tutoring. The following discussion illustrates this point. Consider the following function call in the case of the Lisp Tutor (+ *(cdr list1) (cdr list2)),* where the function *cdr* is used to return the remaining part of the list after stripping the first argument, e.g. applying the function *cdr* to the list ('a b c d') will return the list ('b c d'). Since the ordering of arguments to the function + is not important, the system allows the user to code the two arguments in either order. Thus, when the goal is set to code the first argument, there are two candidate productions, each of which codes *cdr*. When the user types *cdr*, the context is not large enough to make it possible for the system to determine which argument the user is coding. This ambiguity could be resolved only in the next cycle when the next symbol is typed. However, to postpone resolution for a cycle, it would be necessary for the production system to follow both possible branches. That requires matching the user's next step to the subgoal of each production, and thus increasing the amount of pattern matching required.

Moreover, even this simple variability that concerns the typing of the arguments in any order, as long as the ordering is unimportant, becomes very costly when the number of arguments grows larger than two. Currently, the Lisp Tutor easily handles different unimportant orderings of arguments as long as the number of arguments is not greater than two. This requires only two productions to keep track of the two arguments, one checks for the 'list1', in our simple example, and the other for the 'list2'. When the number of arguments is 4, for example, the number of different orderings becomes 4! (24 orderings). This implies

that the production system either has to have four productions, each with 4 matching components, or 24 different productions. In both cases, 24 different matchings are required. In addition, these productions have to follow at least 4 branches at the same time to be able to resolve the ambiguity, which in turn increases the computational cost involved.

Had we decided to represent DISCOVER's knowledge using a rule-based approach, the same problems would have made the attempt to handle larger grain sizes of modeling very difficult to implement. In the case of DISCOVER, only 4 plans are required to check the unimportant ordering of 4 different arguments. For example, consider the following statement *PUT (num1*4) + (num2*3) + (num3*7) + (num4*8) IN newnum*. Since the entire statement is submitted at once, the first plan verifies the existence of the argument *(num1*4)* in the statement, regardless of its order. This is done by making sure that the pattern *(num1*4)* does exist in the statement. The second plan verifies the existence of the pattern *(num2*3),* and so on. Since these patterns are hand coded in the reference solution, only one matching operation is required per plan. For example, the following plan (expressed in POP-11 programming language)

MEMBER(" (num3*7)", statement)

would be enough to make sure that the user has indeed included this argument in his statement. This would have been impossible had DISCOVER allowed the user to enter only one single symbol or token at a time. Of course, the production system could incorporate this approach in its implementation (e.g. requires expected patterns to be hand coded in its rules to reduce the amount of matching components). But then, this would make the system become more or less DISCOVER-like, hand coded reference solution, which in turn would make the system lose its ability to synthesize the solution and simulate the user.

4.4 Immediacy of Feedback

To develop the novice's programming skill, a program diagnosis system must be able to trace the novice's actions and determine when he diverges from a correct solution path so that it can offer suggestions or criticism on individual steps, rather than being limited to advice on complete solution step. By following the novice's actions while trying to put programming concepts together, the system can respond to the underlying misconceptions that motivated the behavior rather being restricted to comments concerning the surface form of the whole solution. This requires, besides model-tracing, support for immediate feedback on both failure and success.

4.4.1 The Need for Flexible Interaction

The principal features of the Lisp Tutor's interaction style can be summarized as follows:

- The system insists that the novice stay on a correct solution path and immediately flags errors. The system reacts to every symbol the novice types and provides immediate feedback as soon as the novice deviates from the solution path.
- The system does not allow the novice to backtrack and delete previously entered code.
- The system uses a menu-based dialogue to track planning decisions and behaviors when it fails to trace them nonintrusively.
- The system forces the novice to enter the code in a left-to-right, top-to-bottom manner. This implies that the next piece of code or the next step on a solution path is decided by the system and not by the novice. Occasionally though, the user is given some freedom in dealing with arguments whose ordering is not important, or even with functions which have the same underlying functionality, such as *cons*, *append* and *list*.

While each of these features has pedagogical justification and close ties to the underlying cognitive modeling, there is no reason why some of these features, especially the first two, cannot be improved to support a more flexible style of tutorial interaction while preserving a close relationship to the model-

tracing approach. Some amount of self-detection and correction of errors may lead to a clearer understanding of the problem and a better explanation of the programming process by the novice and certainly is something that users using the Lisp Tutor have said they wanted (Anderson, 1990). Providing immediate feedback upon every single Lisp symbol is also extremely undesirable and restricting in situations where not enough context has been established for the novice to understand why his solution is wrong.

4.4.2 The Interaction Style of DISCOVER

DISCOVER supports a more flexible style of tutorial interaction that is based on improving the first two features of the Lisp Tutor's interaction style mentioned above. This is achieved by increasing the grain size of automatic tutoring and by providing novices with some opportunity for self-correction of errors. The principal features of DISCOVER's interaction style can be summarized as follows:

- The system reacts to every complete programming statement and expression, not to a single symbol, and provides immediate feedback as soon as the novice wanders off the correct solution path.
- The system supports limited backtracking by allowing novices to delete previously entered code (e.g. parts of the statement currently being completed).
- The system supports an explicit planning mechanism to trace the intentions and high-level goals of the novices. Novices externalize their planning decisions by choosing form a menu of programming concepts rather than through a dialogue.
- The system requires the novice to enter the code in a top-to-bottom manner.

By increasing the grain size of tutoring to a complete statement and expression, DISCOVER provides novices with some opportunity for self-correction and also a larger context for instruction. Since the grain size of tutoring is confined to a single symbol, the Lisp Tutor finds it difficult to explain why a novice's action is wrong at the point which the misconception is first manifested because there is not enough context.

To consider an example, compare a novice who provides *(append x y)* where *(cons x y)* is better for appending the contents of the two lists x and y in a single list returned by the system. It would become easier to explain the choice after the complete statement has been provided rather than after '(append' has been entered. In the case of DISCOVER, this problem does not arise. If the novice provides, for example, 'READ 5 IN num' where 'PUT 5 IN num' is more appropriate, the system explains the choice after the complete statement has been typed in rather than immediately after 'READ' has been selected.

This allows DISCOVER to generate more appropriate explanations and advice that can derive mapping, generalization and coordination that exist between similar programming concepts. For example, in the case of 'READ' instead of 'PUT', DISCOVER informs the novice that it would be better in normal cases where getting an input from the user is not required to use the 'PUT' concept for assigning values to cells. This explanation would not become possible to generate if DISCOVER could not wait to see whether the novice indeed wanted to read 5 and not some other values in cell 'num'.

DISCOVER also supports limited backtracking by allowing novices to delete previously entered parameters and operators of the statement currently being completed. Unfortunately, at present the backtracking is confined to the current statement only. The novice can also cancel the selection of a concept and select a new one that best represents his next goal. For example, if the novice selects the 'READ' concept where 'WHILE' is expected and realizes after completing the selected concept, but before submitting it, that he made an error, he can backspace over the statement. The system would ignore the selection without considering it a deviation from a solution path. This gives the novice some opportunity for self-correction. In fact, there are cases in which the novice may be confused about what goals and plans are appropriate in the current situation and would realize only if he is given a little more time to self-correct. This is not possible with the classical version of the Lisp Tutor.

5. Conclusion

In this paper we have presented an overview of the field of intelligent program diagnosis and critically compared work accomplished on four closely related active diagnosis systems while emphasizing such issues as the representation techniques used to capture the domain knowledge required for the diagnosis, ability to handle the diagnosis of partial code segments of the solutions, features of the user interfaces, and methodologies used in conducting the diagnosis process. Several design considerations to improve the engineering of active approach to intelligent diagnosis were proposed. In particular, we have argued in this paper that the model-tracing based active approach to automatic program diagnosis has several advantages when compared to other approaches. These advantages include (1) the ability for diagnosing very specific errors and providing specific feedback within proper context, (2) the ability to analyze partial solution steps as they are generated by the users and hence reason about the problem solving process, and (3) the ability to minimize the impact of multiple bugs on the diagnosis process. Despite these advantages, we have also argued that this approach to automatic diagnosis tends to be very directive and that some alternative design decisions need to be incorporated into this approach to make it support a more flexible style of user interaction.

Alternatives proposed include (1) supporting an ability to give delayed feedback by increasing the grain size of diagnosis to a complete program statement, not just a single word or token, (2) allowing the users to do limited backtracking by giving them some chance to delete previously entered code and restart, and (3) departing from production system when representing the expertise needed for automatic diagnosis. These alternative design issues were successfully incorporated in the implementation of DISCOVER.

With the rapid growth in the Web development and distance learning technology, we hope to develop a Web version of DISCOVER using some Internet programming languages such as Java to introduce programming concepts and problem-solving to novice users all over the world. The language supported by the system can also vary to cover other languages used in some popular applications such as the scripting language used in the spreadsheets calculations, SQL language used in the database applications, and the HTML language used in the Web applications.

6. References

ADAM, A. and LAURENT, J. 1980. LAURA: A System to Debug Student Programs. *Journal of Artificial Intelligence,* **15:** 75-122.

ANDERSON, J. 1982. Acquisition of Cognitive Skill. *Psychological Review,* **89:** 369-6.

ANDERSON, J. 1990. Cognitive Modeling and Intelligent Tutoring. *Artificial Intelligence and Learning Environments,* Clancy and Soloway (Eds.), MIT/Elsevier.

ANJANEYULU, K. and ANDERSON, J**.** 1992. The Advantages of Data Flow Diagrams for Beginning Programming, In Frasson, Gauthier and McCalla (Eds.), *Intelligent Tutoring Systems,* Springer-Verlag.

BAECKER, R. and DIGIANO, C. 1997. Software Visualization for Debugging. *ACM Communications,* Vol. **40:** 45-54.

BAECKER R. 1990. *Human Factors and Typography for More Readable Programs,* Addison-Wesley, Reading, Mass., 1990.

BARR, A. 1976. The Computer as a Computer Laboratory. *International Journal of Man-Machine Studies,* **8:** 567-596.

BOEHM-DAVIS, D. 1987. Program design languages: how much detail they should include, *International Journal of Man-Machine Studies,* **27:** 337-347.

BONAR, G. 1992. Intelligent Tutoring with Intermediate Representations. *Proceedings of the Second Conference on Intelligent Tutoring Systems (ITS-92),* Canada.

BROWN, M. 1992. ZEUS: A system for algorithm animation and multi-view editing, *Proceedings of the 1991 IEEE Workshop on Visual Languages,* Japan, pp. 4-9.

CLANCY, W. 1987. Qualitative Student Models. *Annual Review of Computer Science,* **1:** 381-450.

CURTIS, B. 1988. Experimental evaluation of software documentation formats. *Journal of systems and software,* **9:** 1-41.

DAVIES, S. 1992. Cognitive Models of Programming and the Design of Support for Learning. *Proceedings of the NATO Workshop on Cognitive Models and Intelligent Environments for Learning Programming,* Italy.

DU BOULAY, J.B.H., O'SHEA, T. and MONK, J. 1981. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. International Journal *of Man-Machine Studies,* **14:** 237-249.

DYCK, J. 1987. Learning and comprehension of Basic and natural language computer programming by novices, *Ph.D thesis,* University of California, Santa Barbara, USA.

EISENSTADT M. 1993. Software Visualization as a pedagogical tool, *Instructional Science,* Vol. **21:** 335-364.

GILMORE, D. and GREEN, T. 1988. Programming Plans and Programming Expertise. *Journal of Experimental Psychology,* **40A:** 69-92.

GOLDSEIN, I. 1974. *Understanding Simple Picture Programs.* PhD Thesis, MIT.

HASHMER, T. 1983. An Empirically-Based Debugging System for Novice Programmers. *PhD Thesis,* The Opend University, UK.

JOHNSON, W. 1990. Understanding and Debugging Novice Programs. *Artificial Intelligence and Learning Environments,* Clancey and Soloway (Eds.), MIT/Elsevier.

LAWRENCE, A. Empirically evaluating the use of animations to teach algorithms. IEEE *Symposium on Visual Languages,* St. Louis.

LIEBERMAN, H. 1984. Seeing what your programs are doing. *International Journal of Man-Machine Studies,* **21:** 311-331.

LUKEY, F. 1980. Understanding and Debugging Programs. *International Journal of Man-Machine Studies,* **12:** 42-71.

MAYER, R. 1981. The Psychology of How Novices Learn Computer Programming. *Computing Surveys,* **13:** 121-141.

MAYER, R. 1985. Learning in complex domains: a cognitive analysis of computer programming, in *Psychology of learning and motivation,* Bower (Ed.), **19:** 89-130, Academic Press, USA.

MILLER, J. 1982. Intelligent Tutoring for Programming Tasks. *Technical Report,* Texas Instruments.

MUKHERJEA, S. and STASKO, J. 1994. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger, ACM TOCHI, Vol. **1:** No. 3 (Sept. 1994), Pages 215-244.

MURRAY, W. 1986. Automatic Program Debugging for Intelligent Tutoring Systems. *PhD Thesis,* Texas University, Austin, 1986.

NWANA, H. 1991. An Approach to Developing Intelligent Tutors. *Proceedings of the 6th International PEG Conference on Knowledge Based Environments for Teaching and Learning,* Italy.

PRICE, B. A. 1993. Principled taxonomy of software visualization. *Journal of Visual Languages and Compuling,* **4**(3)**:** 211-266.

RAMADHAN, H. 1992a. Intelligent vs. Unintelligent Programming Systems for Novices. *Proceedings of the IEEE 15th International Conference on Computer Applications and Systems, USA.*

RAMADHAN, H and DU BOULAY, B. 1992b. Programming Environments for Novices. du Boulay and Lemut (Eds.), *Cognitive Models* and Intelligent Environments for *Learning* programming, Spring Verlag.

RAMADHAN, H. 1997. Improving the Engineering of Model-Tracing Based Approach to Intelligent Program Diagnosis and Tutoring, *IEE Journal of Software Engineering,* **144(3):** 149-161.

RAMADHAN, H. 1997. Model tracing based approach to intelligent program diagnosis. *SQU Journal of Science & Technology,* **2:** 65-76.

RAMADHAN, H. 1999a. Active vs. Passive Systems for Automatic Program Diagnosis, Proceedings of International Conference on HCI (HCI'99), pp 345-351, Munich, Germany.

RAMADHAN, H. 1999b. Improving the Engineering of Immediate Feedback for Model-Tracing Based Program Diagnosis, Proceedings of International Conference on HCI (HCI'99), pp 352-358, Munich, Germany.

RAMADHAN, H. 2000a. DISCOVER: An intelligent system for discovery programming , Journal of Cybernetics & Systems, Vol. **31:** 87-114.

RAMADHAN, H., BRUSILOVSKY, P., and DEEK, F. 2000b. Incorporating Software Visualization in the Design of Intelligent Diagnosis Systems for User Programming, Journal of Artificial Intelligence Review, **01:** 1-24.

REISER, B. 1992. Making Process Visible: Scaffolding Learning with Reasoning-Congruent Representations. *Proceedings of the 2nd Conference on Intelligent Tutoring Systems (ITS '92),* Montreal.

REPENNING, A. 1996. Domain-Oriented Design Environments: Making Learning a Part of Life, *Communications of the ACM***, 9:** 56-72.

RICH, E. 1986. Characterization of plan-based program analysis approaches to debugging. *Personal Communication*, April, 1986.

RUTH, G. 1973. *Analysis of Algortihm Implementations.* Technical Report, MIT Project MAC TR 130, MIT.

SHAPIRO, E. 1983. *Algortihmic Program Debugging.* MIT Press, MIT.

SOLOWAY, E. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering,* September, 1984.

STASKO J. and PATTERSON C. 1992. Understanding and Characterizing Software Visualization Systems, *Proceedings of the 1992 IEEE Workshop on Visual Languages,* USA, pp. 3-10.

SPOHRER, J. 1985. A Goal/plan Analysis of Buggy Pascal Programs. *Human Computer Interaction,* **1**(2)**:** 163:207.

VESSEY, I. 1985. Expertise in debugging computer programs: a process analysis. *International Journal of Man-Machine Studies,* **23:** 459-494.

WEGNER, E. 1987. *Artificial Intelligence and Tutoring Systems,* Morgan Kaufmann Publishers, USA.

Received 22 September 1999
Accepted 3 July 2000