Science and Technology, 2 (1997) 29-38
© 1997 Sultan Qaboos University

Model-Tracing Based Approach to Program Diagnosis

Haider Ali Ramadhan

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

المعالجة الذكية للبرامج باستخدام طريقة المقارنة مع الحلول المثلى

حيدر بن علي رمضان

خلاصة إن إحدى الأساليب المستخدمة في أنظمة البرمجة الذكية لاكتشاف الأخطاء المنطقية في مقارنة خطوات المبرمج أثناء كتابته للبرامج مع خطوات البرنامج النموذجي أو شبه النموذجي . وعندما يكتشف النظام الذكي اختلافا بين الخطوات الصحيحة المذكورة في البرنامج النموذجي وبين خطوات المبرمج يقوم بالإشارة إلى هذه الاختلافات . ويقدم النصحية المناسبة للمبرمج تمكنه من فهم الخطأ . إن هذا الأسلوب يمكن أنظمة البرمجة الذكية من اكتشاف أخطاء دقيقة قد تدل على وجود فهم خاطىء عند المبرمج حول أسلوب حل المسائل الدقيقة ، ومن ثم إعطاء الإرشاد المناسب لتصحيح هذا الفهم . كما إن الأسلوب المذكور يخفف من صعوبة عملية تصميم أنظمة البرمجة الذكية ، وهندسة البيانات المستخدمة لتشغيل هذه الأنظمة . إن أهم عيوب هذه الأنظمة يكمن في إنها تحصر المبرمج في اختيارات ضيقة ، ولا تتيح له المجال لاكتشاف الخطأ بنفسه حيث إنها تتدخل بمجرد اكتشاف الخطأ . إن هذا البحث يتناول طرق وأساليب جديدة للتغلب على عيوب هذه الأنظمة الذكية في مجال البرمجة باستخدام الأسلوب المذكور أعلاه.

ABSTRACT: Model-tracing based approach to intelligent program diagnosis and tutoring emphasizes the fact that the student's design decisions are traced as the student develops the program. Systems based on this model monitor the user's actions as he moves along the solution path, automatically analyzes partial solutions for semantic errors and misconceptions, and offers guidance whenever he deviates from the correct solution path. 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. 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. However, this approach is very directive and interventionist. The user is highly constrained in the solutions that can be developed, since he must conform to the task decomposition and coding sequence enforced by model-tracing systems. This paper critically looks at model-tracing and suggests several solutions and guidelines for bypassing the shortcomings associated with the approach.

Intelligent program diagnosis and tutoring 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 Diagnosis* while the later perform *In-event Diagnosis*.

Systems using post-event diagnosis can be further divided according to their methods of isolating and localizing errors into (1) those using specification based analysis, such as Proust (Johnson, 1990), (2) those using trace-based analysis, such as PDS6 (Shapiro, 1983), (3) those using I/O based analysis, such as BIP (Barr, 1976), and (4) those using model-answer based analysis, such as Talus (Murray, 1986).

Systems using in-event diagnosis 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 diagnosis 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. Generally speaking, these systems rely on some sort of pre-stored requirements for a complete solution, and hence are normally classified as model-answer based systems.

On the other hand, systems using active analysis perform automatic diagnosis by implementing model-tracing (Anderson, 1982). 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. On 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. Example of such systems include GREATERP (Anderson, 1990) (also known as Lisp Tutor), GIL (Reiser, 1992) and DISCOVER (Ramadhan, 1992). These systems tend to be directive and constrain the user in their own-developed solutions. However, through rich interaction and flexible immediate feedback, these systems detect very specific bugs and misconceptions.

Model-tracing based diagnosis

Model-tracing, used in the Lisp Tutor (Anderson, 1982) and GIL (Reiser, 1992), 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 is what 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 work on model-tracing and immediate feedback (Anderson, 1990) 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 (1990) 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.

Problems with the model-tracing approach

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-down 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 (Wenger, 1987; Nwana, 1991).

Second, the success of model-tracing based systems depends heavily on the extent of their model-tracing knowledge; this includes the number of correct rules in the ideal user model and the number of the mal-rules in the buggy user model (Nwana, 1991). For example, the production system of the Lisp Tutor currently contains more than 1,200 rules, more than half of which are mal-rules. Production systems, despite their many advantages, impose several computational problems when utilized to

support model-tracing based diagnosis and tutoring, especially of large problems (Anderson, 1990).

Third, the important programming activity of debugging is taken away from the user since model-tracing based systems, in principle, do not permit floundering (Wenger, 1987; Nwana, 1991). 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 (see (Ramadhan, 1992) for a detailed description of the DISCOVER's visual environment). 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.

An overview of the system DISCOVER

The aim of the DISCOVER system is to teach the novice programmer how to compose and coordinate programming concepts and statements to solve given programming problems under the guidance of the intelligent programming expert, and thus build effective problem-solving skill. The user has to build the solution to the problem presented by properly putting together programming concepts from the menu. Once a concept is completed and accepted by the syntax directed editor, it is passed to the diagnosis expert for automatic analysis. 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).

Much like Talus and Bridge (Bonar, 1992), DISCOVER relies on a prestored 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. Unlike these systems, however, DISCOVER like the Lisp Tutor is capable of interactively analyzing partial solution code and providing immediate feedback on both success and failure. By doing that, DISCOVER explicitly guides the novice in the process of putting together programming concepts to solve the given problem. DISCOVER monitors the novice's actions, not on a symbol-by-symbol basis like the Lisp Tutor and GIL, 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 the Lisp Tutor and GIL, however, DISCOVER (1) utilizes goals and plans (not a production system) to represent the knowledge of its domain expertise, (2) does not keep an account, at least currently, of common error patterns (the buggy model), (3) supports an explicit planning mechanism during all stages of the problem-solving process to trace the user's intentions (e.g. by selecting the concepts from the menu during problem-solving, the user non-intrusively tells the system his intentions and goals), and (4) supports an ability to give delayed feedback by increasing the grain size of automatic diagnosis 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.

The programming language of DISCOVER is a pseudo-code based, algorithm-like language. The language is kept simple in order to reduce the number of abstract programming ideas and concepts. The language omits advanced concepts such as recursion and procedures, avoids having too many programming tricks to be learned, and avoids requiring the learning of low-level syntax details. Programming concepts supported currently include *create, put, read in, write out, while-endwhile, if-istrue-isfalse*. The interface, as shown in figure 1, appears to the user as a collection of eight windows. The four windows on the left side of the interface, namely the *memory space, inputspace, output space*, and the *algorithm space*, represent the model of the underlying notional machine.

Despite these positive aspects of DISCOVER, the system has several weaknesses when compared against other model-tracing based systems (discussed in more detail in the sections to follow). Briefly, the main weaknesses of the system include:

1. The system has no explicit representation of knowledge. DISCOVER utilizes a hand-coded reference solution to conduct automatic diagnosis.
2. The reference solution is represented in terms of a Proust-like goal-and-plan tree. However, these goals carry no structure and are used for keeping track of various programming concepts selected by the user during the problem solving task.
3. The plans are used as a tool for coding and matching syntactic templates that correspond to the low-level code objects in DISCOVER's language, hence plans in this context are language dependent and reflect the actual implementation and not the possible implementation of the underlying knowledge of novices.
4. The current representation of the reference solution limits DISCOVER's ability to handle large programs. Since DISCOVER must assume a small number of goals and plans in order to be able to predict what the

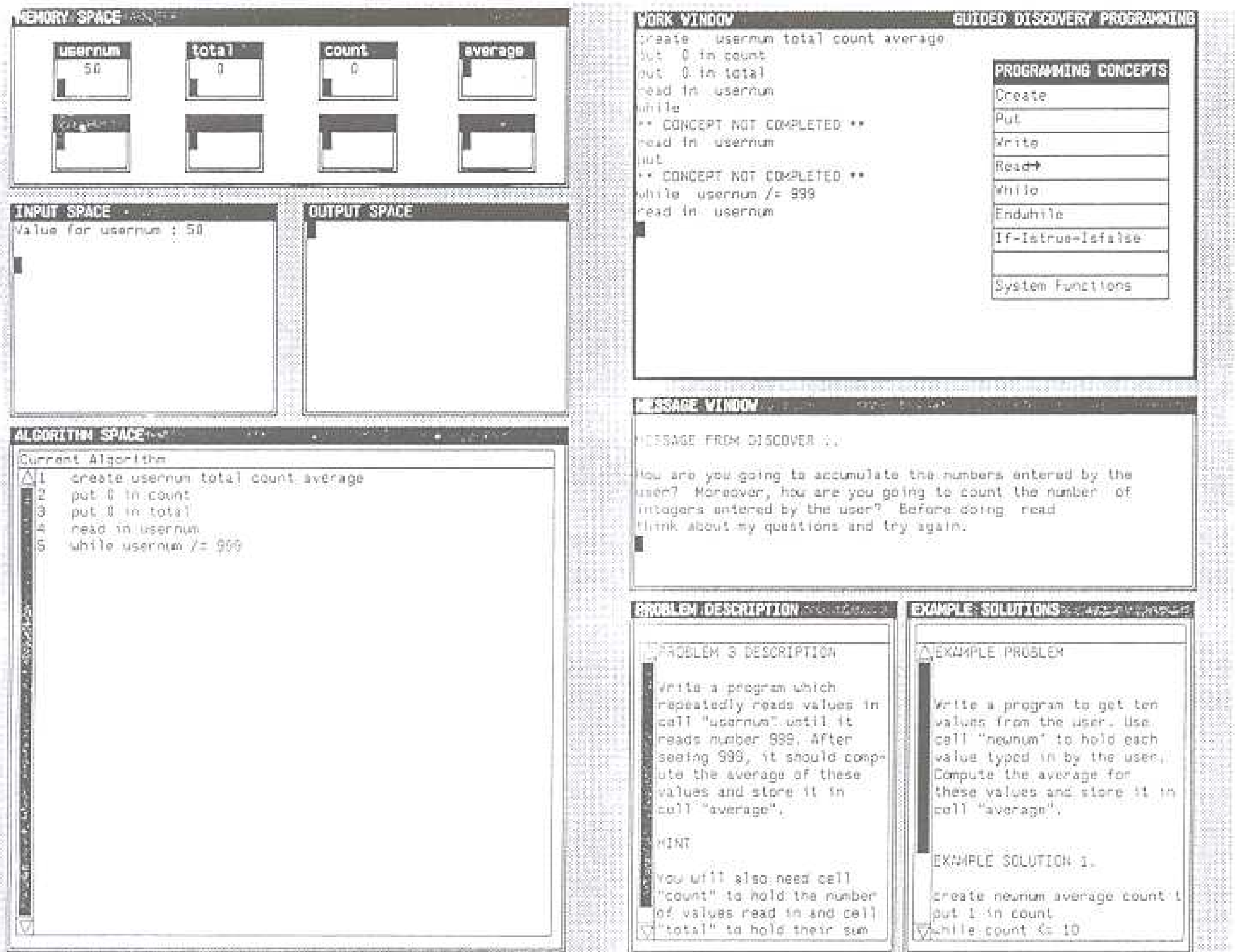

Figure 1. DISCOVER's interface.

novice's current state might be at any given time during problem-solving process, it can handle only those programs which achieve a relatively small number of goals. To support larger problems would certainly increase the bushiness of the reference solution trees, and both the time and effort needed to implement.

DISCOVER's 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 gives 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.

An improved version 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. However, before stating these principles, two issues are worth noting here. **First**, a number of empirical studies have shown that novices normally tend to follow a top-down approach when learning to program and debug, and quite infrequently tackle their problems using a bottom-up approach, even when it is supported by the system.

It is worthwhile to clarify the use of the term 'top-down' in this paper, since it has another usage in the programming literature. For the purpose of this research, the term 'top-down' simply expresses the fact that novice programmers tend to use a depth-first approach when trying to comprehend the behavior of their programs and when trying to solve programming problems. During the comprehension process, novices tend to read the program statements in their physical order to gain understanding of parts of the program (Nanja, 1988). Novices start with the first statement in the program and move all the way down to the last statement in the program, emphasizing the physical order in which these statements are written. In other words, novices tend to read the program from the beginning to end like a piece of prose. This is in marked contrast to the expert programmers who tend to read the program in the order it would be executed, thus emphasizing the dynamic order in which the statements are executed .

During problem-solving, novices tend to write their programs statement-by-statement, moving on to the next statement only when the current statement is completed. Even at the statement level, novices tend to complete the statement in a left-to-right order, i.e. moving in a forward reasoning direction (Reiser, 1992). In short, the term 'top-down' implies here that novices tend to write their programs in a depth-first, left-to-right, forward reasoning direction. In a way, this corresponds to the procedural and textual decomposition of the program, and *NOT* to the stepwise refinement model used by the expert programmers and software engineers for decomposing a task from high-level specifications into more elementary levels.

The **second** issue concerning model-tracing is that the ideal system needs to carefully monitor interaction with the user. When designing intelligent programming systems for novices, the emphasis should not be given only to the final product (a complete program) but also to the very process of programming itself through which the user has to go to come up with the final solution. This requires the system to support a model-tracing based approach to diagnosis.

The issue being tackled here is not whether model-tracing and immediate feedback provide a useful diagnosis methodology nor whether a top-down strategy is appropriate for novices nor even whether in-event debugging systems are better than post-event systems. The issue is how we can improve 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. This 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). This 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 (Bonar, 1992).
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.

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 (Spohrer, 1985). For each goal, there are a 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.

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 make up what is known to be the production system. A production system 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 to implement model-tracing (discussed in the next section). These systems have to consider 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 (Anderson, 1990).

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 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 forces 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

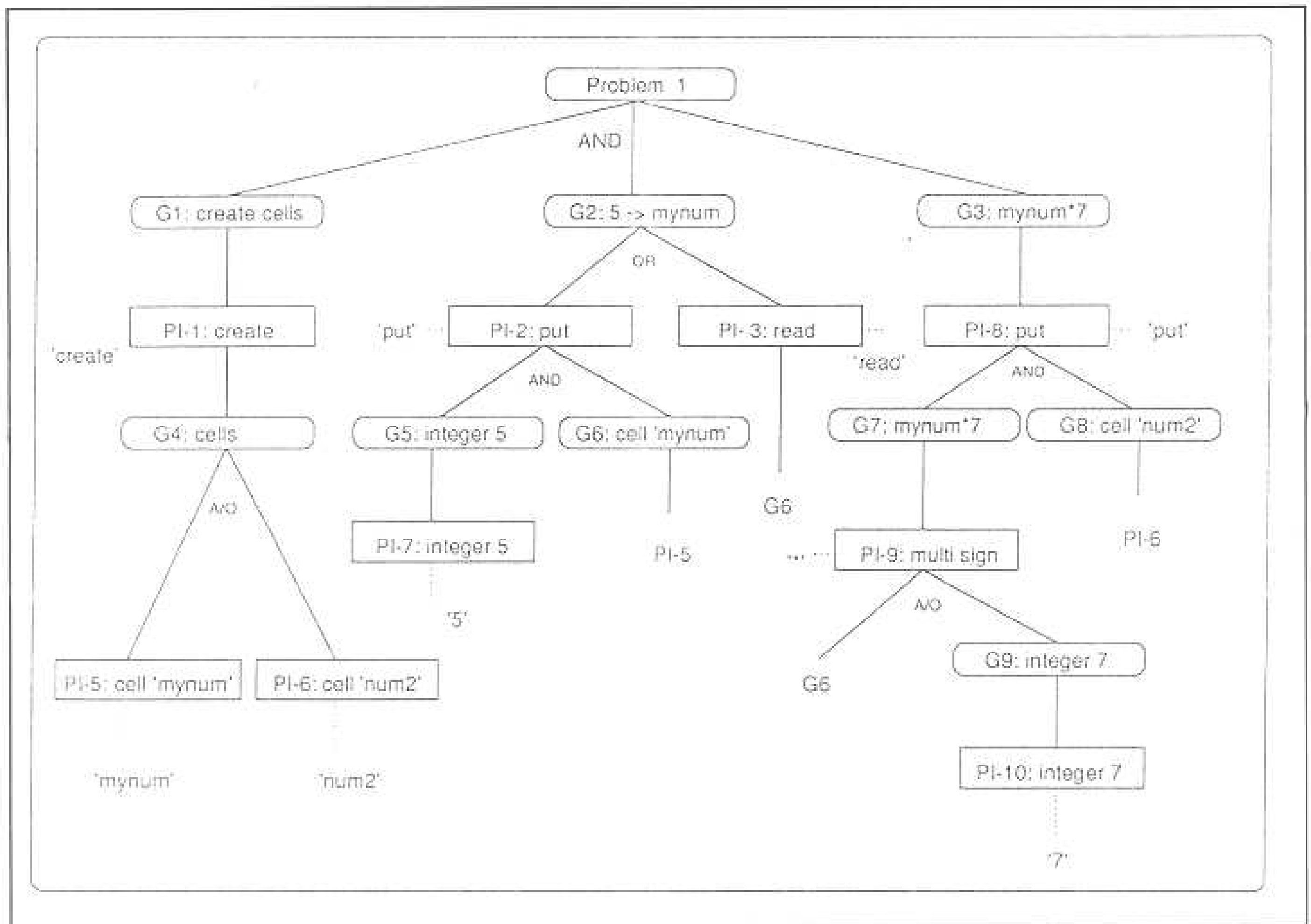

Figure 2. An example of DISCOVER's reference solution.

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. Figure 2 shows an example of a reference solution for a simple problem, which creates cells 'num2' and 'mynum', stores the number 5 in cell 'mynum', and stores the result of multiplying the content of cell 'mynum' by 7 in cell 'num2'.

As shown, 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. Note how variability as well as constraints over the user solution are represented using AND/OR clauses. 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.

RULES VS. PLANS: It is not easy for a rule-based system to allow the type of variability supported by DISCOVER, while at the same time handle 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))*. 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, this 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 user has indeed included this argument in his statement. This would have been impossible had DISCOVER allowed the user to only enter 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.

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 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 (see (Anderson, 1990) for a more detailed discussion on this principle).

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-down 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. 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.

DISCOVER'S INTERACTION STYLE: 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 of 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 from a menu of programming concepts rather than through a dialogue.
- The system requires the novice to enter the code in a top-down manner.

By increasing the grain size of tutoring to a complete statement and expression, DISCOVER provides novices 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 (list x)y)' where 'cons x y)' is better. 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 represents best 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. Besides supporting larger grain size of tutoring and backtracking, DISCOVER also gives immediate feedback on success.

Conclusion

An intelligent program diagnosis system using model-tracing based approach 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. However, this approach tends to be very directive and interventionist. The user is highly constrained in the solutions that can be developed, since he must conform

to the task decomposition and coding sequence enforced by model-tracing systems. To bypass the pitfalls associated with traditional model-tracing approach as it was implemented in Lisp Tutor and GIL, model-tracing systems need 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. This 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). This 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 (Bonar, 1992).
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.

References

ANDERSON, J., 1982. Acquisition of Cognitive Skill. *Psychological Review*, **89**, 369-406.

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

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

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

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

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

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

NANJA, M., 1988. *An Investigation of the On-Line Debugging Process of Expert and Novice Student Programmers*. PhD Thesis, Oregon State University.

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, 1991.

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

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

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

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

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

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

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

Received 8 February 1996
Accepted 24 February 1997