MENO-II: AN INTELLIGENT TUTORING SYSTEM FOR NOVICE PROGRAMMERS *

Elliot M. Soloway, Beverly Woolf,
Eric Rubin, and Paul Barth

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

We report here on the goals and status of an intelligent tutoring system being developed for novice Pascal programmers. We also describe our approach to program understanding, bug finding, and the inference of misconceptions. Examples of the system's capabilities are given.

Introduction

We all know — and grumble — about the error messages that one gets from a computer system. That problem is magnified for the novice programmer who is learning a first programming language. With the number of individuals who are learning programming rapidly growing, there is a clear need for intelligent tutoring systems which can assist the novice at a most critical time: when he/she is alone, one-on-one, with the beast.

Objectives and Status of System

We are building an intelligent tutoring system, Meno-II, tailored to the needs of novice Pascal programmers. The goals of Meno-II *are:*

* catch run-time (semantic and pragmatic) bugs in the student programs **
  - focus on introductory programs: straight line, branching, simple looping.
  - must be fast and robust; at least 85% of student programs are buggy
* suggest misconceptions in the students' heads which underlie the bugs
  - "talk" in language which is close to programming so student will understand
* instruct/tutor the student with respect to the misconceptions.

The current status of the system corresponding to these objectives is:

* Meno-II can find over two dozen bugs with respect to repetition and assignment
* Meno-II can suggest underlying misconceptions for a subset of those bugs
* tutoring has not yet been implemented in Meno-II.

Leverage on Program Understanding: The Bug Finder

The Bug Finder must be able to recognize two types of bugs: problem independent ones, and problem dependent ones. An example of the former type is the explicit inclusion of an increment to the index variable in a for loop. Bugs of this sort reflect confusion about the semantics of the various programming language constructs.

In order to recognize problem dependent bugs, however, the Bug Finder must be told what the program is supposed to do; oftentimes a student program will run, but it will solve the "wrong" problem. For example, in Figure 1a, we depict a problem .for which the the program in Figure 1b is the correct solution; the program in Figure 1c does not solve the problem, but does in fact execute. In other intelligent tutoring systems, the student provided a specificiation of the goals of the program (e.g., [3]). In contrast, our approach is to have the teacher provide the specification, since everyone in the course will be working on the same problem.

The vehicle for this "intention specification" is a Problem Description Language (PDL) which we are developing for introductory programming problems. The PDL allows us to specify the components of a program solution to the intended problem. In Figure 2. the PDL solution for the problem in Figure 1 is given.

This language is based on what an expert programmer might know about problems of this type. In particular, this knowledge is focussed on types of looping plans and the various roles which variables play in programs. For example, the

* This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant No. MDA903-80-C-O508. Any opinions, findings, conclusions or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

•• While interesting, we are not concerning ourselves with syntax errors (see [11]).

program in Figure 1b illustrates what we call the New_Value Controlled Running_Total Loop Plan; it is just a special case of the Running__Total Loop Plan, in which the loop, while accumulating a total, is controlled by the value of the read variable (i.e., the New__Value Variable). This knowledge is described at greater length in [8]. This characterization of programming knowledge draws on the work of [33, [53, [63, and [113.

The Bug Finder is given a PDL specification of the program solution; currently this is in a procedural form, but eventually it will be table driven. The Bug Finder then parses as a student program into a syntax tree representation. In order to facilitate speedy retrieval, this structure has been augmented with: (1) extra links (e.g., all occurrences of a variable are chained together), (2) semantic annotations at a node which summarize the subtree below (e.g., the statement i:=i+1 would be summarized as "increment-by-one"). The PDL solution description is then matched against this augmented parse tree. For example, in comparing the description in Figure 2 with the program in Figure 1c, the Bug Finder recognizes that the New_Value Variable, Score, should have been modified in the loop by a read, as opposed to being modified like a counter. Bugs are then reported to the Tutor Module (described below).

The guiding metaphor of the Bug Finder is pattern matching at the loop plan level. Thus, two different programs, which solve the same problem can be recognized as such. The Bug Finder keys off the loop being used and calls in knowledge specialized to that particular loop to help in the matching. Currently, the Bug Finder does not do exhaustive control or data flow analysis; it does do spatial data flow analysis to recognize that:

```
X := Sum;
Sum := X + Nu_Value;
```

is equivalent to Sum := Sum + Nu__Value. Nor do we map the progam into a logic formalism and do the analysis in this latter representation; we feel that without extensive translation the explanation that would result from this type of approach would be "to far away" from the student's understanding of the program to be of much help.

Tying Bugs to Misconceptions: Building a Model of the Student

Given a bug in the program, one needs to infer the underlying misconceptions, in the head of the student, which could have caused the bug ([23). Based on the expert's plan knowledge we have built a knowledge network, the Competence Model, which describes the correct components of a loop and their relationships. We also have built [a] Bug Model which specifies the surface characteristics in the program of the common bugs which we have identified ([73, [83). Finally, a set of inference routines tie the Bug Model to the Competence Model in order to derive the Performance Model, i.e., a model of the student's possible misconceptions. Both the Bug Model and the Competence Model are implemented in a version of KL-ONE ([1]) which we have augmented with some special arcs to facilitate quick retrieval.

For example, the bug in the program in Figure 1c could have been caused by the following misconceptions, which would be inferred by our Tutor Module: (1) the student simply misunderstood the problem, (2) the student did not understand how to get new, successive values, or (3) the student did not understand how the read statement worked. For example, the student might have thought that the variable Score was declared to be of "type read" and that if 1 were to be added to it, then the system "would know" to do a read.

Currently, the Competence Model contains about 200 concepts which represent loop plans, variable roles, and the various relationships among these entities. The Bug Model contains approximately 50 bugs associated primarily with the improper initialization and modification of various variables. We are expanding both networks, and we are developing rules which can better diagnose misconceptions (e.g., if there are multiple, related bugs, then this should help focus the diagnosis). Still to be worked in is the history of the student's interactions.

Tutoring

The current system functions as a "smart compiler"; it finds bugs, diagnoses misconceptions, and simply lists this analysis to the student. We plan to have the Tutor engage in a dialogue with the student for better diagnosis and instruction purposes. However, we do not plan on allowing the user to communicate via natural language; rather, we plan to use dynamically generated multiple choice questions presented via a menu screen. We will, however, allow the user to submit program fragments in response to questions from the system. With this system, we want to explore the range of possible interaction scenarios. In particular, our objective is to "parameterize" the amount and type of interaction in order to instruct along the continuum from tutoring ([93) to coaching ([4]).

Concluding Remarks

There are a number of assumptions which underlie our approach, and which are still open questions. For example:

1. Can we define a PDL sufficient for the problems under consideration? We feel that this will be answered in the affirmative.

2. Will the pattern matching approach, without exhaustive data or control flow analysis be sufficient to handle the looping problems? Can this be extended to handle programs with procedures? Can we continue heuristically handling questions that typically required rigorous deduction systems? These are the the most troublesome questions.

3. Will our analysis, in terms of plans and variable roles, provide enough leverage into the students' thinking? Since we do not see the history of the design, as Miller's SPADE-0 system does ([5]), it is unclear whether we can sufficiently reconstruct the student's plan from simply the raw code.

These are empirical questions which we plan to explore through tests of the system. In the fall semester of 1981, we plan to try out Meno-II in a class of 700 students taking introductory Pascal programming. They certainly will exercise the system, and tax it to its limits.

References

[1] Brachman, R. (1981) "KL-ONE Reference Manual", BBN Report, Cambridge, Mass.

[2] Brown, J.S.. Burton, R.R. (1978) "Diagnostic Models for Procedural Bugs in Mathematics," Cognitive Science, June.

[3] Goldstein, I. (1974) "Understanding Simple Picture Programs," Technical Report AI-TR-29**, M.I.T. A.I. Lab. Cambridge.

[4] Goldstein, I.P. (1976) "The Computer as Coach: An Athletic Paradigm for Intellectual Education". Artificial Intelligence Lab., Memo 389, M.I.T.

[5] Miller, M.L. (1978) "A Structured Planning and Debugging Environment for Elementary Programming," Int. J. Man-Machine Studies, 11. PP. 79-95.

[6] Rich, C, and Shrobe. H. (1978) "Initial Report on a Lisp Programmer's Apprentice," IEEE Trans, on Software Engineering, Vol. SE-4, No. 6.

[7] Soloway, E., Bonar, J., Ehrlich, K. (1981) "Cognitive Factors in Programming: An Empirical Study of Looping Constructs", COINS Technical Report 81-10, Univ. of Mass.. Amherst.

[8] Soloway, E., Bonar, J.. Woolf. B., Barth, P., Rubin. E.. Ehrlich, K. (1981) "Cognition and Programming: Why Your Students Write Those Crazy Programs," Proc. of the National Educational Computing Conference, Texas.

[9] Stevens, A., Collins, A. (1977) "The Goal Structure of a Socratic Tutor". Proc. *of* ACM Annual Conference.

[10] Teitelbaum, T. and Reps, T. (1980) "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Department of Computer Science, Cornell University, Technical Report 80-421, May.

[11] Waters, R.C. (1979) "A Method for Analyzing Loop Programs," IEEE Trans. on Software Engineering. SE5:3. May.

Problem. Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the correct average. That is, is should not count the final 99999.

Figure 1a *— A Simple Looping Problem

```
program Student6_Problem3;
        var Count, Sum, Number : Integer; Average
  real ;
        begin
        Count := 0;
        Sum := 0;
        Read (Number);
        while Number <> 99999 do
                begin
                Sum =.r Sum * Number;
                Count : = Count + 1 ;
                Read (Number)
                end;
        Average := Sum / Count;
        WriteIn (Average)
        end.
```

Figure 1b — A Correct Solution

```
program Student 17_Problem3;
        var N, Sum, Score : Integer; Mean : real;
        begin
        N := 0;
        Sum := 0;
        read (Score);
        while Score <> 99999 do
                begin
                Sum := Sum + Score;
                Score := Score + 1;
                N := N + 1
                end;
        Mean := Sum / N;
        WriteIn ('the mean = ', Mean)
        end.
```

Figure 1c — An Executable, But Incorrect Solution

```
Test Variable is the New_Value Variable
New_Value Variable is the Read Variable
there is a Counter Variable
there is a Running_Total Variable
Average :=
   Running Total Variable/Counter Variable
```

Figure *2*

Informal PDL Specifcation of Problem in Figure 1a