INTRODUCTION: MOTIVATIONS AND GOALS

For the past few years our group has worked on systems for automatic diagnosis of non-syntactic errors in novices' computer programs (Soloway, et al., 1984; Johnson, 1986; Sack, 1989). Such errors (or "bugs") go beyond simple mistakes in the language syntax; they reflect deeper misunderstandings of how an algorithm works and how it should be implemented in that language. Our automatic debuggers are on-line help facilities designed to find such errors in students' programs. To be effective, an automatic program debugger must have methods allowing it to both:

• **Identify** errors in computer programs, and

• **Explain** to the student the errors it has found.

Like many other problems tackled by researchers in Artificial Intelligence (AI), the problems of bug identification and bug explanation are open ended: no parsimonious, general solutions exist for them. Our scientific aesthetic motivates us to search for economical, non-idiosyncratic solutions to the problems of bug identification and explanation, but the methods we use to find such solutions look more like those of engineering than of science. We iteratively refine our solutions by constructing working programs that incorporate our current ideas about bug identification and explanation. We submit our working systems to measures of performance, by trying them out with real students. Finally, we build a successor that we hope will measure up better than the previous debugger.

This paper is an attempt to make explicit some of the evaluative decision making that goes into the patching and redesign of our automatic debuggers. Specifically, this paper is a description of the performance tests to which we submitted the instructional debugger, PROUST, the results of those tests, and a discussion of what we believe are the design implications of the test results. We want to describe how CHIRON (our newest debugger) was born out of the ashes of the earlier PROUST.

The approach we follow has many precedents in the AI literature. R1 (McDermott, 1982) became new-XCON (Bachant and Soloway, 1989); EMYCIN (Clancey, 1982) was distilled out of MYCIN (Shortliffe, 1976) as the Truth Maintenance System (Doyle, 1978) was isolated from EL (Stallman and Sussman, 1977); GPS (Newell and Simon, 1972) begat STRIPS (Fikes and Nilsson, 1971) begat ABSTRIPS (Sacerdoti, 1974) begat NOAH, (Sacerdoti, 1975); and SOPHIE I grew into SOPHIE II and then SOPHIE III (Brown, Burton, and de Kleer, 1982). Typically, improvements included in a sequel system are responses to either anecdotal evidence that the old design needs fixing (e.g., "Spade-0 does not use information in the problem description. This severely restricts its range of planning advice and debugging assistance ... To address this difficulty, we have designed and are now implementing Spade-1, an improved
environment, ..." (Miller, 1982, p.130). ) or to common sense arguments that the old design is fundamentally flawed, [e.g., PAM (Wilensky, 1978) was developed after SAM (Cullingford, 1978). "Since people can understand situations that they have not experienced, they must have an understanding ability beyond that obtainable from simple script or frame application." (Wilensky, 1981, p. 136). Anecdotal evidence and common sense arguments are necessary ammunition to attack a previous AI design, but we think more is needed to make substantial improvements to an Intelligent Tutoring System (ITS) like our last debugger, PROUST. The acid test of an ITS is "Can it teach students what it is supposed to teach them?" An ITS stands or falls upon its ability to perform in the face of the day-to-day, pragmatic challenges of the classroom. Consequently, the feedback central to developing an ITS is data that documents its performance in instructing students.

We begin our critique of PROUST by asking, "Can PROUST teach students what it's supposed to teach them?" More specifically:

1. Does PROUST globally improve programming performance on an exam?
2. Does PROUST, as an on-line aid, help students to find and correct bugs?
3. Does correcting bugs in homework improve subsequent ability to correct bugs?

We continue by describing an investigation designed to measure PROUST's bug-finding skills, addressing the question:

4. Can PROUST perform the task it teaches? How accurate is PROUST in identifying errors in student computer programs?

In this chapter, our tests of PROUST are sandwiched between an introductory section on what PROUST is and how it works and a wrap-up section on our newest debugger CHIRON. CHIRON embodies some (as yet unevaluated) answers to questions that have appeared as a result of tests performed on PROUST. In the conclusion section we summarize the benefits we have gathered from testing PROUST.

DESCRIPTION OF PROUST

What PROUST Does

PROUST has been used by novice Pascal programmers as an on-line debugging aid. Normally, students use a text editor to create their programs and then invoke PROUST by typing a command like

```
proust <program filename> <output filename>
```

from their top-level environment (e.g. cshell in UNIX). PROUST analyzes the student program, and then writes into an output file specified by the student a description of the errors found. The student then returns to the text editor to make changes in the program, and is able to invoke PROUST again.
PROUST first checks the student program for syntax errors. If it finds any, PROUST simply informs the student that there are syntax errors and terminates its analysis, leaving the student to correct errors of this kind. Only when the syntax is correct does PROUST begin its main analysis. Currently, PROUST analyzes programs written to solve two programming assignments, including the Rainfall Problem shown in Figure 1. All data examined in this paper comes from this problem.

Write a Pascal program that will prompt the user to input numbers from the terminal. Each input stands for the amount of rainfall in New Haven for a day. Note: since rainfall cannot be negative, the program should reject negative input. Your program should compute the following statistics from this data:
1. the average rainfall per day
2. the number of rainy days
3. the number of valid inputs (excluding any invalid data that might have been read in)
4. the maximum amount of rain that fell on any one day.

The program should read until the user types 99999. This is a sentinel value signaling the end of the input. Do not include the 99999 in the calculations. Assume that if the input value is non-negative and not equal to 99999, then it is valid input data.

Figure 1: The Rainfall Problem.

The Rainfall Problem requires students to generate a moderately complex program of between 30 and 50 lines. Mental execution of a program for this problem is not trivial, nor are the data flow relations. The student must also remember a myriad of small details (such as whether a test should use the predicate <and> or the predicate <or>).

This description of PROUST's performance is organized around a sample buggy program segment (Figure 2) written for the Rainfall Problem, PROUST's resulting bug report (Figure 3), and code correcting these bugs (Figure 4).

This buggy program segment in Figure 2 follows a segment in which a user's input is accepted and VALIDDAYS, TOTALRAINFALL, and MAXIMUM, are computed. The following three bugs occur in the program segment:

- **Missing Divide-By-Zero Guard**: If the user inputs no valid days, the average (which is the total amount of rain that fell divided by the number of valid days input) should not be computed because the computation will result in a divide-by-zero error.
- **Missing Output Guard on Average**: Similarly, if the average is not calculated it should not be output.
- **Missing Output Guard on Maximum**: If the user entered zero valid days no maximum amount of rainfall that fell on some given day can be computed. Therefore, the maximum should not be output.
Intelligent Tutor Design as Iterative Engineering

{01} 

{02} AVERAGE := TOTALRAINFALL/VALIDDAYS; (*Missing Divide-By-Zero Guard*)

{03} Writeln('Average = ', AVERAGE); (*Missing Output Guard on Average*)

{04} Writeln('Maximum = ', MAXIMUM); (*Missing Output Guard on Maximum*)

{05} ...

Figure 2: Sample buggy program segment.

After identifying the three bugs shown in Figure 2, PROUST outputs the bug report shown in Figure 3.

Starting bug analysis, please wait

NOW BEGINNING BUG REPORT

>>> Now Reporting CRITICAL Bugs in the OUTPUT part of your program

BUG 1: You need a test to check that at least one valid data point has been input before line 2 is executed. The average will bomb when there is no input.

>>> Now Reporting MINOR Bugs in the OUTPUT part of your program

BUG 2: The average is undefined if there is no input. But line 3 outputs it anyway. You should output the average only when there is something to compute the average of.

BUG 3: The maximum is undefined if there is not input. But line 4 outputs it anyway. You should output the maximum only when there is something to compute the maximum of.

BUG REPORT NOW COMPLETE

Figure 3. PROUST bug report.

All three of these bugs can be fixed with one correction: surrounding the three lines in the original program segment with one appropriate guard (see Figure 4). Nevertheless, PROUST reports them as three errors. The reasons are first that PROUST finds each of them separately: Bug 1 is found when PROUST is checking to see how the average is computed, Bug 2 is found when PROUST is checking to see how the average is output, and Bug 3 is found when PROUST is checking to see how the maximum is output. Second, PROUST does not correct bugs for

9/28/09
students. Rather, it is up to the student to figure out that all three bugs can be eliminated with one guard.

```plaintext
01 ...
02 If (VALIDDAYS > 0)
03     Then Begin
04         AVERAGE := TOTALRAINFALL/VALIDDAYS;
05         Writeln('Average = ', AVERAGE);
06         Writeln('Maximum = ', MAXIMUM);
07     End;
08 ...
```

**Figure 4:** Program segment correcting the bugs identified in PROUST's report.

As we shall see, interactions with PROUST like that shown in Figure 3 do help students to find bugs, and to become more effective debuggers. However, the nature of the interaction is almost certainly not optimal, and we will suggest ways in which it might be improved.

### How PROUST Works

PROUST's internal libraries contain four kinds of information, illustrated in Figure 5-9. *Problem specifications* (Figure 5) describe the objects in the problem and the goals that constitute a problem solution. *Goals* (Figure 6) reflect standardized parts of program code. *Plans* (Figure 7) are stereotypical methods for realizing typical programming goals. Several alternate plans may provide alternate means of implementing a single goal.

*Plan-difference rules* relate PROUST's plans to possible plan variants in students' code. These rules (a type of production rule) have a test pattern that is matched against the student's code and an action that transforms the student's code into a plan in PROUST's library. Plan-difference rules include *transformation rules* (Figure 8) which match correct plan variants and *bug rules* (Figure 9) which match incorrect plan variants.

In all of these figures, PROUST's code is in normal type, with explanatory comments in italics.
Rainfall is the problem. The resulting program must process a sequence of inputs, DailyRain, that are of the ObjectClass ScalarMeasurement.

**DefObject**

?Rainfall:DailyRain ObjectClass ScalarMeasurement

Input a sequence of values and terminate when the sentinel value 99999 is read in.

**DefGoal** Sentinel-Controlled Input Sequence

( ?Rainfall:DailyRain, 99999 )

Make sure that none of the values read in are less than zero.

**DefGoal** Loop Input Validation

( ?Rainfall:DailyRain, ?Rainfall:DailyRain<0 )

Calculate the average of the values read in, and output the computed average.

**DefGoal** Output ( Average (?Rainfall:DailyRain) )

Count the number of values read in, and output the count.

**DefGoal** Output ( Count (?Rainfall:DailyRain) )

Count the number of values read in that were greater than zero, and output the count.

**DefGoal** Output ( Guarded Count

( ?Rainfall:DailyRain, ?Rainfall:DailyRain>0 )

Find the maximum value read in, and output the maximum.

**DefGoal** Output ( Maximum (?Rainfall:DailyRain) )

**Figure 5**: A PROUST problem specification
A count goal has two main parameters: the sequence of values being counted, and the variable that holds the current value of the count.

**Form:** Count ( ?New, ?Count)

The variable that holds the current value of the sequence being counted is referred to as ?New.

**SequenceVariable:** ?New

The variable that holds the current value of the count is referred to as ?Count.

**ResultVariable:** ?Count

The central part of a counter plan is its Update: segment.

**MainSegment:** Update:

A count goal should be called a "counter" when generating comments to be read by students.

**NamePhrase:** counter

A count goal can be fulfilled with either of two plans:
1. A Counter Plan, which initializes ?Count, and then updates it every time the next value of ?New is encountered, or
2. A Split Counter Plan, which separates the values to be counted into two or more sets (e.g. values greater than zero and values less than or equal to zero), counts the sets separately, and then adds the results together to get the final value of ?Count.

**Instances:** Counter Plan
SplitCounter Plan

Figure 6: A PROUST goal.

One variable is used in the counter plan; it is called ?Count.

**Variables:** ?Count

A counter plan’s template has two parts to it:
1. the initialization sets the value of ?Count to zero, and
2. the update increments the value of ?Count by 1 every time the next value is encountered of the sequence of values being counted.

**Template:**
Init: ?Count := 0
Update: ?Count := (?Count + 1)

Figure 7: A PROUST plan.
Figure 8: A PROUST transformation rule.

Figure 9: A PROUST bug rule.

PROUST uses the following steps to identify bugs in a student's program.

1. PROUST is given a problem specification and the name of the file containing a student's Pascal program. After PROUST has scanned the problem specification to determine which goals should be implemented in the student program, it selects an initial goal set to consider, and gives it to the interpretation manager.

2. The interpretation manager accepts the initial goal set, orders them, and puts them onto the goal agenda. The first goal on the agenda is selected, and PROUST fetches all plans that implement this goal. This list of plans is given to the plan matcher.
3. The plan matcher tries to match plans on the list against the code in the student program stopping with a successful match. A plan usually contains many segments, and is matched segment-by-segment. Plans also can contain sub-goals. If the plan matcher encounters a plan with subgoal, it adds the subgoals of the plan to the goal agenda and works on matching the subgoals before returning to work on the plan that spawned the subgoals. If a segment of a plan cannot be matched then that plan match is suspended. If the current plan is suspended, PROUST tries to match the next plan on the list for the current goal, until a successful match is found. If one or more plans are matched successfully, then processing proceeds with step 5. Otherwise all of the suspended plans are processed by the plan difference rules. A suspended plan can be reactivated if the unmatched segment can be matched using plan difference rules.

4. Plan difference rules seek to identify the student code as a variant (correct or buggy) of a plan that PROUST knows. If the plan difference rules are able to explain all of the current match errors for a given plan, then that plan is sent back to the plan matcher. For example, a split counter plan for the rainfall problem, counts separately the zero-rainfall days.

5. If a plan is matched successfully to the student code, i.e., if all of the segments of a plan have been matched (with or without the aid of plan difference rules), then PROUST stores it (along with the goal it implements) in the current interpretation for the student program. If more than one plan matched successfully, the interpretation manager applies some heuristics to select which one is the best plan. If the current plan is not a successful match, PROUST tries to match the next plan on the list for the current goal, until a successful match is found.

6. The next goal is then selected by the interpretation manager PROUST works to successfully match a plan for this goal.

7. At the end of an analysis, the interpretation assessor assigns one of the following confidence ratings:
   In a full interpretation every piece of student code matches a programming plan.
   In a partial interpretation some of the code matches PROUST's plans. PROUST outputs comments on each line of code it did not understand.
   In an incomplete interpretation, most of the student code does not match PROUST's programming plans, and there is no coherent analysis of what the student is trying to do.

8. The bug reporter generates a bug report based on the interpretation and the confidence rating.
Implementation

PROUST is 15,000 lines in length and is written in T, Yale's dialect of LISP. It takes about 0.5 - 1.0 minute to execute and runs on a DEC VAX 11/750 with 4 megabytes of memory. PROUST's knowledge base contains about 37 goals, 55 plans, and 70 bug rules.

CAN PROUST TEACH WHAT IT'S SUPPOSED TO TEACH?

Having described the workings of PROUST, we turn now to the central purpose of this paper—to examine PROUST's performance and what we can learn from it. We examined performance (students' and PROUST's) on the three bugs shown in Figure 2. These bugs have the following desirable properties for answering our questions:

High-Frequency: we needed to find bugs that were made by a sufficiently large number of students to allow us to carry out statistical tests. Moreover, we wanted bugs that were seen both in the homework assignments and on the midterm. These constraints significantly limited the bugs that we could use in our analyses: the three bugs in Figure 2 are essentially the only ones that met these constraints. Of the 80 students in the study, 60 (75%) of them made one or more of these three Missing Guard bugs.

Representativeness: The Missing Divide-By-Zero Guard is what we call a common bug. No programming knowledge is required to avoid or find this bug because underlying knowledge comes from basic algebra: a divisor must never be zero. In contrast Missing Output Guard on Average and Missing Output Guard on Maximum are what we call idiosyncratic bugs because finding and correcting them requires knowledge specific to programming: it is a generally accepted programming convention that variables are only output if they hold meaningful information. In the Rainfall problem, the variables VALIDDAYS and MAXIMUM hold meaningful information only if there has been some user input of valid rainfall values. The Missing Output Guards protect against outputting these variables unless they actually contain meaningful information.

We do not claim that the distinction between common and idiosyncratic bugs constitutes any great leap forward in student modelling; nevertheless, it is a common-sense, knowledge-based, student-oriented distinction that we, at the time, felt would enhance our understanding of the data. Since the data analyses for this paper were performed members of our group have made much progress in the areas of student errors and student modelling in the programming domain (Spohrer, 1989) and bug explanation and tutorial dialog in the programming domain (Littman, 1989).

With respect to these bugs, we examined PROUST's educational effectiveness by asking (a) whether PROUST globally improve programming performance on an exam, (b) whether PROUST helps students to find and correct bugs, and (c) whether correcting bugs in homework improves subsequent ability to correct bugs. The following sections describe experiments addressing these questions.
Does PROUST globally improve programming performance on an exam?

To address this question, we designed a midterm exam, which was taken about one week after the Rainfall homework problem, and which required students to identify and repair bugs in three programs.

Student Groups
In the spring of 1985 at Yale University, the following two sections of the introductory Pascal programming course intended for humanities students were created by random assignment:
ACCESS Group (39 students) As they were developing their solution to the Rainfall Problem (Figure 1), students in this section had access to PROUST, and thus could run PROUST and receive PROUST's analysis of the programs they wrote for their homework assignments. They used PROUST on the Rainfall problem only.
NO ACCESS Group (41 students): Students in this section did not have access to PROUST. The two sections met at different times, but they both had the same instructor, received the same instruction, used the same textbooks, took the same examinations, and did the same homework assignments.

Results and Discussion
On the midterm examination, the mean score of the ACCESS Group was higher than that of the NO ACCESS Group. Controlling for SAT scores in the two groups, an analysis of covariance showed that 16% (Baker, private communication) of that difference was accountable for by PROUST. Though this performance difference did not quite attain conventional levels of statistical significance, a single exposure to PROUST had an observable effect on students programming skills tested one week later in a different setting.
In analyzing the effect of an instructional innovation like PROUST, one would like to take into account all of the possible variables that might influence the result. Pragmatically, it is impossible to enumerate, much less gather information about all of the variables that might effect the overall result. (Background education in mathematics and computers seem relevant, but so do "small" details like the percentage of the hours of classroom lecture that a given student attended prior to working on the Rainfall Problem.)
In our analysis, we only took into account the SAT scores of the two groups. However, the SAT scores played a large role in measuring the effects of PROUST: without taking SAT scores into account the ACCESS group did much better than the NO ACCESS group. This observation indicates the crucial importance of considering the setting in which an instructional tool is used.
We could have made PROUST look considerably better had we omitted this consideration. If our purpose were simply to give a global evaluation of PROUST's educational benefits, we could stop here. The answer to "Can PROUST teach debugging?" seems to be "yes," (if not a statistically resounding YES!!) under demanding conditions (one week delay, different problems, and limited exposure to PROUST), and with serious attention to differences in student abilities (SAT scores).
The following sections explore the nature of learning with PROUST.
Does PROUST help students find and correct bugs?

Does PROUST, as an on-line aid, help students identify and repair bugs? To answer this question, we collected copies of each syntactically correct version of a student's program, along with the output provided by PROUST for those in the ACCESS Group.

Student Groups

GOOD ADVICE group: This group is a subset of students with access to PROUST identified as follows. We examined the programs collected, together with any output given from PROUST. For each of the three bugs we identified a group of students who (a) introduced the bug into their programs, and (b) turned on PROUST and got correct advice at least once.

NO ADVICE group: These are the students in the section without access to PROUST. Because the GOOD ADVICE group is a non-random subset of the ACCESS group, we checked the comparability of the GOOD- and NO-ADVICE groups with respect to SAT scores and quality of the homework programs as ranked on a 3 point scale by two expert programming tutors. The experts achieved an 85.4% agreement rate on all programs. There were no significant differences on either measure.

For each of these groups, we categorized in the following way what the student did about each missing-guard bug appearing in their program:

- FIX: The bug was correctly fixed in the last version of the program.
- NO FIX: The bug still appeared in the last version of the program.

Results

Figure 10 shows that students with good advice from PROUST were more likely to fix the two output guard bugs (see Figure 2) than were students with no advice from PROUST. But there was no such difference for the Divide-by-zero bug.

<table>
<thead>
<tr>
<th>Bug (Guard Omitted)</th>
<th>Divide - by - zero</th>
<th>Output , average</th>
<th>Output, maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug Fixed:</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Advice:</td>
<td>Good</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>(p &lt; 0.24)</td>
<td></td>
<td>(p &lt; 0.05)</td>
<td>(p &lt; 0.01)</td>
</tr>
</tbody>
</table>

Figure 10: For each bug, the number of students with good or no advice who ultimately fixed, or did not fix, the bug in the Rainfall homework problem. (Statistical significance was calculated by Fisher's exact p test.)
The differential effectiveness across bugs corresponds to our distinction between bugs correctable by common knowledge (missing divide-by-zero guard), and bugs requiring programming knowledge (the missing-output guards). Novice programmers are not familiar with programming conventions. Consequently, they can't identify bugs based on such knowledge until they are explicitly told the relevant information, here that a variable should not be output unless it holds meaningful information. PROUST can explicitly point out this programming convention to students as it does in Figure 3.

Discussion
The results of this analysis are not clear cut recommendations for modifying PROUST, nor do they make transparent the cognitive activities of the students. Nevertheless, these results do cast some light on students' and PROUST's (in)abilities to provide students with on-line debugging help. The fact that PROUST was not very useful to students in helping them fix common bugs could be interpreted in at least three different ways

*With respect to PROUST's construction:* PROUST's knowledge about domains other than programming is shallow. Perhaps PROUST's common-knowledge should be increased to allow PROUST to better explain common bugs to students.

*With respect to students' cognitive make-up:* On the other hand, perhaps, filling out PROUST's common knowledge wouldn't improve PROUST's performance in explaining common bugs to students. Perhaps, since knowledge like "The number 0 should never be used as a divisor" is so ubiquitous, students may tend to ignore simple repetitions of this information. If this is the case, then PROUST's advice with respect to common bugs would only be useful if PROUST were able to state its advice in a novel manner.

*With respect to PROUST's communication ability:* PROUST might produce better results for common bugs if it more directly stated the relevant common knowledge. PROUST's bug report in Figure 3 does not, for example, explicitly state that the number 0 should never be used as a divisor.

Does correcting bugs in homework improve subsequent ability to correct bugs?
As part of their midterm examination students were asked to identify and repair the bugs in programs. Points were awarded separately for identification and repair. One of the buggy programs was structurally identical to the Rainfall Problem. Specifically, the Rainfall Problem required the processing (average, maximum, etc.) of integers that represented inches of rainfall per day, and the midterm exam program required the processing of integers that represented insurance claims. This program was seeded with the three missing-guard bugs shown in Figure 2. Thus, we were able to compare how well students did at fixing bugs in their own programs with how well students did at fixing the same bugs in the program on the midterm.

Student Groups
The students included in this analysis had made all three MISSING GUARD bugs in their own Rainfall programs. The students classified as FIXED ALL were those who fixed all three of the MISSING GUARD bugs in their own Rainfall programs. The remaining students were classified
as DID NOT FIX ALL. We found no significant differences between the two groups with respect to their mean SAT scores. Here we drop distinctions between the nature of PROUST advice (accurate and inaccurate). Maintaining this distinction along with the FIX-ALL/DID-NOT-FIX-ALL distinction would have made our sample sizes too small for reasonable analyses. Because of this grouping, the following results indicate the effect of successful bug-fixing experience on later ability to identify and repair similar bugs, but they do not directly indicate effects of PROUST.

Results
On the midterm, there were 6 possible points awarded for correctly identifying all three MISSING GUARD bugs and 4 awarded for correctly repairing them. Figure 11 summarizes the performance of the two student groups. Although t-test comparison of the identification scores for the FIX ALL and NOT FIX ALL student groups did not quite attain the conventional 0.05 level of significance for identification, there an indication that those students who fix all three bugs are superior at bug identification. As shown in Figure 11, those students who made and fixed all three bugs in their own programs did score significantly higher on bug repair on the midterm than those who did not fix all bugs.

<table>
<thead>
<tr>
<th>MidTerm:</th>
<th>IDENTIFY</th>
<th>REPAIR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>mean</td>
</tr>
<tr>
<td>Homework:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Fix All</td>
<td>26</td>
<td>2.8</td>
</tr>
<tr>
<td>Fix All</td>
<td>8</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>t = 1.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.05 &lt; p &lt; 0.10)</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11**: Mean number of points awarded for identification and repair of MISSING GUARD bugs on the midterm exam for students who did and did not fix all bugs on their homework. (p values are for one-tailed t tests)

Discussion
In sum, students in the FIX ALL group are better able both to identify and to repair the missing-guard bugs on the midterm. On the one hand these results are not counter intuitive: one would expect that those who do their homework will score better on tests. On the other hand, however, what is interesting and rather non-intuitive about these findings is their particularity. We have pinpointed exactly what homework (fixing all three missing-guard bugs) correlates with a better score on a specific part of a midterm exam (the part that contains the missing-guard bugs). This micro analysis allows us to identify specific bugs where PROUST’s identification assistance is most helpful.
Summary of PROUST's Educational Effectiveness

Students who had access to PROUST's bug identification reports while doing programming assignments performed better on a midterm examination specifically designed to exercise bug identification and repair skills. This macro effect is all the more impressive because the improvement in midterm performance is due entirely to using PROUST on a single homework assignment.

Students who had the benefit of accurate bug reports from PROUST were significantly better at fixing bugs on their programming assignments than students who did not have such reports. Students who succeeded in fixing particular bugs on their programming assignments (with or without accurate advice from PROUST) were significantly better at repairing similar bugs on the midterm examination. Thus debugging help such as that provided by PROUST could potentially produce improved debugging ability.

CAN PROUST PERFORM THE TASK IT TEACHES?

How accurate is PROUST in identifying bugs in computer programs? Program debugging is still more of an art than a science. Our purpose in building automatic debuggers has been in the interest of helping and instructing students in the process of debugging. However, by building automatic debuggers we have also had to carefully define what we think the process and skills of debugging are irrespective of how debugging can be taught to students. PROUST's debugging knowledge and skills are its domain intelligence, what Anderson (1988) calls "...the backbone of any ITS." The importance of this domain intelligence, the skills and knowledge of debugging, and its current status as an art make it a research area unto itself.

In this section we make explicit some of the strengths and weaknesses of the model of debugging embodied in PROUST. We hope that this analysis will uncover some of the knowledge and processes that must be coded into any effective automatic program debugger. And, we hope that this analysis of PROUST's debugging skills will give us better insight into what skills novice programmers must be taught in order to be able to debug programs.

Experimental Design and Description of the Data

To quantify PROUST's debugging behavior we tested PROUST using Rainfall programs written by 39 students in an introductory programming course offered at Yale in the spring of 1985. This programming course is offered every year at Yale. Student programs from this course in 1982 and 1983 were used as training data during PROUST's construction. Johnson (1985) tested PROUST's accuracy on student programs written for this course in 1984.

As test data, we used the first syntactically correct program version written by each student. These first versions usually contain more bugs than do subsequent versions, and so are a more rigorous test of PROUST's diagnostic power. The student programs were collected by modifying the operating system so that every version of a program saved by the student is copied and saved into a permanent file. We also counted the number of versions of the program each student saved and counted the number of bugs in both the first syntactically correct version and
the last version saved by the student (i.e., the version that the student handed in for a grade). None of these 39 programs were seen by the author of PROUST (Johnson, 1986). Four expert graders independently identified the bugs in the student programs. They met to discuss their independent results, work out disagreements, and finally agree upon one set of results. The bugs listed in Figure 12 and discussed below were identified by this procedure. The diversity of these data on which we tested PROUST is considerable. PROUST's average run time to diagnose bugs in the first program versions ranged between 10 and 60 seconds, with a mean of 33.3 sec. The number of bugs in students' first programs ranged from 1 to 14, with a mean of 6.1 and from 0 to 9 with a mean of 2.8 in the final version. The students' persistence in pursuing the assignment to completion also We also counted the number of versions of the program each student saved and counted the number of bugs in both the first syntactically correct version and the last version saved by the student (i.e., the version that the student handed in for a grade). There were 67 unique bug types, and 236 instances of bug occurrence. Figure 12 shows in how many initial programs each of the 67 unique bug types appeared. A few bugs types accounted for the vast majority of the bug instances. The five most common bugs account for 116, or about half of the bug occurrences. Thirty four bug types appear in just one program.

<table>
<thead>
<tr>
<th>Number of Programs</th>
<th>Bug Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>missing guard on average output</td>
</tr>
<tr>
<td>28</td>
<td>missing guard on maximum output</td>
</tr>
<tr>
<td>21</td>
<td>missing guard on average calculation</td>
</tr>
<tr>
<td>13</td>
<td>malformed loop termination test</td>
</tr>
<tr>
<td>12</td>
<td>missing re-input inside of loop</td>
</tr>
<tr>
<td>12</td>
<td>missing guard after rainfall is re-input</td>
</tr>
<tr>
<td>6</td>
<td>missing maximum initialization</td>
</tr>
<tr>
<td>6</td>
<td>missing maximum output</td>
</tr>
<tr>
<td>5</td>
<td>malformed average calculation</td>
</tr>
<tr>
<td>5</td>
<td>missing rainfall input initialization</td>
</tr>
<tr>
<td>5</td>
<td>missing maximum update</td>
</tr>
<tr>
<td>4</td>
<td>missing rainy day initialization</td>
</tr>
<tr>
<td>4</td>
<td>missing rainy day update</td>
</tr>
<tr>
<td>4</td>
<td>missing total rainfall initialization</td>
</tr>
<tr>
<td>4</td>
<td>missing day count initialization</td>
</tr>
<tr>
<td>3</td>
<td>average declared integer</td>
</tr>
<tr>
<td>3</td>
<td>&quot;DIV&quot; used in average calculation instead of &quot;/&quot;</td>
</tr>
<tr>
<td>3</td>
<td>malformed total rainfall update statement</td>
</tr>
<tr>
<td>3</td>
<td>missing day count output</td>
</tr>
<tr>
<td>3</td>
<td>spurious average initialization</td>
</tr>
<tr>
<td>3</td>
<td>spurious guard on day count update</td>
</tr>
<tr>
<td>3</td>
<td>spurious initialization</td>
</tr>
<tr>
<td>1-2</td>
<td>(38 other bugs)</td>
</tr>
</tbody>
</table>

**Figure 12:** Most common bug types, and number of programs in which each was found.
PROUST's Accuracy in Identifying Bugs

To evaluate PROUST's ability to identify bugs in students' programs we compared PROUST's bug diagnoses with those of expert human coders. Out of the 236 bug instances found by human coders in 39 programs, PROUST found 100 (42.3%), missed the remaining 136 (57.7%), and produced 38 false-alarm bugs (i.e. inaccurately identified correct code as buggy code.). 42.3% seems like a surprisingly low rate for recognizing bugs, especially since we know that PROUST is able to improve students' debugging. In the following sections we provide some micro-analyses of this result in order to better understand what this 42.3% accuracy rate means.

When is PROUST inaccurate?

We explained previously how, after completing an interpretation of a student program, PROUST rates its interpretation according to how confident it is of the interpretation's validity. A full interpretation occurs when PROUST is able to match every piece of code to a programming plan. A partial interpretation occurs when PROUST is only able to match some of the code to plans. PROUST's output after a partial interpretation consists of a set of comments about the code it did understand plus a listing of the lines of code it did not understand. PROUST labels an interpretation incomplete when it hasn't matched enough of the code to produce a coherent analysis of what the student is trying to do.

Figure 13 shows, for each level of interpretation, the number bugs found by PROUST and the human coders, the number found by the human coders and missed by PROUST, and the number of false bugs PROUST "found," which were not judged bugs by the human coders.

<table>
<thead>
<tr>
<th>Interpretation:</th>
<th>Bugs Identified</th>
<th>Bugs Missed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True Bugs</td>
<td>False Bugs</td>
</tr>
<tr>
<td>Full (16 programs)</td>
<td>69</td>
<td>13</td>
</tr>
<tr>
<td>Partial (14 programs)</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>Incomplete (9 programs)</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Found by Human Coders</td>
<td>100</td>
<td>38</td>
</tr>
</tbody>
</table>

Figure 13: Bugs identified and missed by PROUST and by human coders.

These results show that PROUST is usually correct when it thinks that it is correct (i.e. when PROUST rates its interpretation full). This micro-analysis points out a feature of PROUST that should be incorporated into our next debugger: the debugger should rate its confidence in an interpretation so that students can accept the debugger's advice with a grain of salt if the debugger is less than certain about its interpretation of a given student's program.
Could tweaking PROUST make it a lot more accurate?

We noticed a few cases in which very minor glitches with PROUST resulted in exceedingly poor performance. Consequently, we were attracted by the idea that perhaps, in principle, PROUST was very accurate at correctly identifying bugs in student programs. Perhaps, if PROUST could be tweaked a little it would show its true colors. We discovered this hypothesis to have some truth to it, reanalyzing the same data as used above.

We made four minor revisions to PROUST. The revisions made resulted in improved accuracy performance by PROUST in 12 of the 39 student programs. To give the reader an idea of just how minor these revisions were we will explain one in detail. The specific revision we will discuss accounted for all of PROUST's increased performance in 4 of these 12 student programs. PROUST's difficulty with these four programs came from its inability to recognize the stopping value in the main loop of the program (a solution to the Rainfall Problem should stop asking for new data after the user has entered the sentinel value 99999). Only one very weak bug rule was in PROUST's bug library to cover cases in which students used the wrong sentinel value as the stop condition for the main loop. Specifically, the bug rule that PROUST had would check to see if the sentinel value used was 9999 if PROUST couldn't find exactly that expected value. In the 4 programs PROUST had problems with the students had used 999999 as the sentinel value. To fix PROUST we simply beefed-up the bug rule so that it could handle 9, 99, 999, 9999, 999999, 9999999, etc. This allowed PROUST to identify the 999999s as buggy sentinel values. After this and other similarly small fixes, we re-ran PROUST on the same data. The result was almost a 20% increase in PROUST's accuracy: after tweaking, PROUST was able to correctly identify 61.4% of the bugs in students' programs. Figure 14 shows the bug counts for the initial and and tweaked versions of PROUST for our data and for that of Johnson (1985).

<table>
<thead>
<tr>
<th>Data Source:</th>
<th>Sack &amp; Soloway</th>
<th>Johnson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proust Version:</td>
<td>Initial</td>
<td>Tweaked</td>
</tr>
<tr>
<td>Bugs Identified:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>True Bugs</td>
<td>100</td>
<td>146</td>
</tr>
<tr>
<td>False Bugs</td>
<td>38</td>
<td>59</td>
</tr>
<tr>
<td>Bugs Missed:</td>
<td>136</td>
<td>91</td>
</tr>
<tr>
<td>Found by Human Coders</td>
<td>236</td>
<td>378</td>
</tr>
</tbody>
</table>

Figure 14: Bugs identified and missed by the initial and tweaked PROUST in two studies.

Although tweaking did not make PROUST perfect, it did dramatically improve its accuracy rating. It is difficult to interpret the implications of this result. One would hope that, with increased usage, small glitches (like the one described above) would get worked out and that tweaking would become unnecessary. However, this result does highlight the fact that
PROUST's bug rules can be fragile and idiosyncratic. We have discovered a way of dispensing with bug rules that is summarized later in this chapter when we describe CHIRON. Figure 14 also contains the results Johnson obtained for PROUST before and after his tweaking. Again, we see that Johnson's small revisions (different from ours, but also small) resulted in a dramatic improvement in PROUST's bug identification abilities: from 52% before tweaking to 77% after tweaking.

The likely source of difference between our accuracy figures and those of Johnson seems to be the following: Both our group and Johnson (1985, p.230) noticed that it is difficult for an isolated individual to find all of the bugs in student programs. Several people working together will usually be able to identify all of the bugs. Consequently, an accuracy rating based upon a single person's bug count will be more optimistic than an accuracy rating based upon a group's bug count, simply because a single person's bug count will document fewer bugs for PROUST to find. We had four expert graders to identify bugs in the student programs, while Johnson worked alone. We found 6.1 bugs/program, while Johnson found about 5.0. If his programs were actually equally buggy, he should have found around 465 errors instead of 378. Using this total, tweaked PROUST found 42%, of the bugs, and tweaked PROUST 62%, results essentially identical with ours.

**How can PROUST's knowledge-base be improved in order to improve its bug identification accuracy?**

The weakness with both of the preceding analyses is that, even though both of them provide us with a more detailed understanding of PROUST's bug identification abilities, neither provides suggestions for making PROUST better at the task of finding bugs. To get at this issue we try to identify specific inabilities of PROUST that hinder it in its task of finding bugs. Specifically, in this micro-analysis we will show one way in which PROUST's knowledge base could be improved.

**PROUST's Lexical Inabilities**

Lexical cues in programs (i.e., the meaning of variable names and output messages) can provide people who read a program with a lot of information about the program author's intentions. For example, consider a statement to use the current day's rainfall, RAIN, to update the value, MAX, of the maximum amount of rainfall that fell on any given day. As a first attempt, we have seen students write

```
MAX := RAIN
```

instead of

```
If (RAIN > MAX) Then MAX := RAIN
```

PROUST's knowledge-base does not contain any lexical information in it, consequently, to PROUST, the assignment

```
MAX := RAIN
```

looks like

```
X7 := Z12
```

where X7 and Z12 might be any meaningless symbols. Our panel of four expert graders looked through the bugs that PROUST had been unable to identify and found that 38 (16%) of the
student bugs missed by PROUST and 12 (32%) of its false alarms could have been avoided through use of lexical information.

Of course, these estimates are only guesses about what PROUST might have been able to do had it known more. But, we believe the observation that an automatic program debugger should be able to utilize lexical knowledge is a good observation. In order to test how feasible it would be to provide an automatic debugger with a useful amount of lexical knowledge, we examined our 39 student programs more carefully. In particular, we wanted to know if a simple mechanism (like pattern matching on commonly occurring variable names) would suffice to improve PROUST’s performance. We went through the student programs to determine two properties of the variable names:

- **Variability**: We wanted to know how many different name instances were associated with each of variables in the Rainfall Problem, and how frequently students used the same name instance.

- **Discriminability**: We wanted to know how often a name chosen by one student to denote a particular variable was used by other students to denote different variables. We use two criteria: 100% (a name used by no other student) and 80% (a name not used by more than 20% of students) to mean something other than its central meaning.

**Results**

What we found (Figure 15) was rather surprising. First, students use a limited range of variable names in the Rainfall Problem, and many students use the same names. Second, if a name is used for one variable, then it is rarely used (by any student) for a different variable. In other words, an automatic debugger equipped with a simple matcher and short lists of the variable names usually associated with particular Rainfall Problem variables, could, in principle, take advantage of the lexical cues contained in variable names.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Number of Students using:</th>
<th>Discriminability:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100% 80%</td>
</tr>
<tr>
<td></td>
<td>most common name</td>
<td>100% 80%</td>
</tr>
<tr>
<td></td>
<td>one of most common two</td>
<td></td>
</tr>
<tr>
<td></td>
<td>names</td>
<td>used</td>
</tr>
<tr>
<td>Total Rainfall</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td>Average</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Day Count</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Maximum</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Rainfall Input</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Rainy Day Count</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Figure 15**: Variable names used in student programs.
Discussion and Extensions
Our analysis of PROUST's lexical inabilities is the clearest illustration we have of PROUST's knowledge deficiencies. We performed other analyses of this sort. For example, our panel of 4 expert graders found that 71 (30%) of the bugs missed by PROUST and 16 (42%) of the false alarms could have been avoided had PROUST been able to perform data and control flow analyses of the sort commonly used in many optimizing compilers (Aho, Sethi, Ullman, 1986)\(^1\). However, these other micro-analyses did not suggest simple additions to PROUST like the simple pattern matching suggested by the lexical micro-analysis. Even though adding a component to PROUST to perform data and control flow analyses would not be difficult, it is difficult to decide when and where PROUST would make use of the flow analyses; and, it is difficult to see exactly how flow analysis knowledge could be inserted into the knowledge representation languages used in PROUST. Questions like these have pushed us to design and build a completely new bug identifier in order to test out our speculations. We will describe our answers to these knowledge-base questions in the upcoming section on CHIRON.

In its interpretation process, PROUST fetches the a goal of the problem specification and then decides which plan, that satisfies the current goal, best describes the manner in which the student has tried to fulfill the current goal in their program. If none of the plans listed under the goal can be matched to the student's code without mismatches and no set of bug rules can explain the mismatches, then PROUST's interpretation process is stopped cold: it cannot proceed to the next goal before finding some interpretation for the current goal. This goal-by-goal approach makes PROUST vulnerable to giving up its interpretation completely if it cannot find a plan for the first goal of the problem specification. Almost all of PROUST's incomplete interpretations occurred when it was unable to match the first goal. Although this result illustrates the fact that PROUST's interpretation process should be more flexible, the result does not reveal what strategy should be implemented to allow the kind of flexibility necessary.

---

\(^1\)The results of a control flow analysis shows the order in which statements will be executed in a program. Data flow analyses allow one to determine which READLN or assignment statement is responsible for a given variable's value in a given segment of a program.
CHIRON: ANSWERS TO QUESTIONS RAISED BY PROUST
In this section we describe the manner in which CHIRON will interact with students. Next, we explain how we think our current and proposed work on CHIRON addresses questions raised in the PROUST evaluation. We are describing a prototype version of CHIRON that has not yet been used by students. Many parts of CHIRON are still under development. For example, we are developing the machinery necessary for CHIRON to not only identify errors, but to fix them as well. Collected together in this section is a summary of our current formulation of CHIRON's bug identification techniques. A more detailed description of the bug identification techniques used in CHIRON can be found in Sack(1989).

What CHIRON does
CHIRON prompts the user for a name of a file to be analyzed, which the student types:

Input the name of the file that you would like analyzed.
<<< Waiting for input >>> "program_filename.pas"

CHIRON first checks the program's syntax. If it is syntactically correct, CHIRON performs control and data flow analyses and then starts its goal and plan analysis. After CHIRON has finished its goal and plan analysis, it outputs a general description of the errors found, and then enters a loop from within which a student can ask questions. The student can terminate interaction with CHIRON by typing quit, return to a text editor, make changes to the program, and then reinvoke CHIRON for further advice.

To illustrate what CHIRON does, Figure 16 shows a buggy solution to the following problem. Figure 17 shows how CHIRON answers questions after it has analyzed this program and Figure 18 lists examples of the types of questions CHIRON can answer.

AVERAGE PROBLEM: Read in numbers until 99999 is input. Prompt the user before each number is read. Calculate and output the average of all the numbers input except the 99999.
```verbatim
{01} PROGRAM Average(INPUT, OUTPUT);
{02} VAR Count: INTEGER;
{03} Sum, New, Avg: REAL;
{04} BEGIN
{05}   Sum := 0;
{06}   Count := 0;
{07}   WRITELN('Input a number');
{08}   READLN(New);
{09}   WHILE (New <> 99999) DO BEGIN
{10}     Sum := Sum + New;
{11}     Count := Count - 1;
{12}       (* Bug 1: counter augmented by "-" not "+" *)
{13}     Count := Count - 1;
{14}       (* Bug 2: counter augmented twice *)
{15}   END;
{16}   IF (Count > 0) THEN BEGIN
{17}     Avg := Sum / Count;
{18}     WRITELN('The average is ', Avg);
{19}   END;
{20} END
```

**Figure 16:** A buggy solution to the averaging problem.
A COUNTER is a plan that accepts a sequence of values and keeps track of the number of values in that sequence. The sequence of values counted is usually generated by an ENUMERATOR plan (see documentation for ENUMERATOR plan for more details). A COUNTER has two parts and five relations. Some of the relations specify how the parts of the COUNTER coordinate with the parts of the ENUMERATOR.

part1) INIT usually initializes the count to 0.

part2) UPDATE increments the count by 1 every time a new value is seen.

relation1) INIT->UPDATE specifies that there is data flow from the INIT to the UPDATE.

relation2) INIT->ENUM_UPDATE specifies that the counter is initialized before the sequence of values is updated (by the ENUMERATOR).

relation3) UPDATE->UPDATE specifies that there is data flow from the UPDATE back to the UPDATE.

relation4) ENUM_INIT->UPDATE specifies that the initial value, of the sequence of values being counted, is not overwritten before the count is UPDATED.

relation5) ENUM_UPDATE->UPDATE specifies that the current value, of the sequence of values being counted, is not overwritten before the count is UPDATED.

1 <<< Waiting for input >>> What’s a COUNTER?

A COUNTER is a plan that accepts a sequence of values and keeps track of the number of values in that sequence. The sequence of values counted is usually generated by an ENUMERATOR plan (see documentation for ENUMERATOR plan for more details). A COUNTER has two parts and five relations. Some of the relations specify how the parts of the COUNTER coordinate with the parts of the ENUMERATOR.

part1) INIT usually initializes the count to 0.

part2) UPDATE increments the count by 1 every time a new value is seen.

relation1) INIT->UPDATE specifies that there is data flow from the INIT to the UPDATE.

relation2) INIT->ENUM_UPDATE specifies that the counter is initialized before the sequence of values is updated (by the ENUMERATOR).

relation3) UPDATE->UPDATE specifies that there is data flow from the UPDATE back to the UPDATE.

relation4) ENUM_INIT->UPDATE specifies that the initial value, of the sequence of values being counted, is not overwritten before the count is UPDATED.

relation5) ENUM_UPDATE->UPDATE specifies that the current value, of the sequence of values being counted, is not overwritten before the count is UPDATED.

2 <<< Waiting for input >>> What’s wrong with my COUNTER?

Your COUNTER does not work correctly, specifically your COUNTER UPDATE does not work correctly, specifically I expected to find a +, but I was only able to find a ADDITIVE_OPERATOR.

3 <<< Waiting for input >>> Explain what an ADDITIVE_OPERATOR is.

An ADDITIVE_OPERATOR is an operator that accepts two numbers and combines them in some way. + and - are ADDITIVE_OPERATORS.

4 <<< Waiting for input >>> I want to quit so that I can fix my program.

GoodBye!

Figure 17: Dialog with CHIRON about the buggy average program.
Goal / Plan Information

Terminology Questions
   Explain what a counter is.
   Explain what a counter update is.

Implementation Questions
   How is the counter implemented in my program?
   How is the counter update implemented in my program?

Error Questions
   What is wrong with my counter?
   What is wrong with the counter update?

Control / Data Flow Information

Control Flow Questions
   Does control flow from the assignment statement on line 8 to the writeln on 25?
   How does control flow from line 8 to line 25?
   Why doesn't control flow from the assignment on line 8 to the writeln on 25?

Data Flow Questions
   Is there data flow from the assignment statement on line 8 to the writeln on line 25?
   Does the value given to X in the assignment on line 8 survive between the assignment on line 10 and the writeln on line 25?
   How does data flow from line 8 to line 25?
   Why doesn't data flow from line 8 to line 25?

Low Level Data Flow Questions
   What is the [GEN or KILL or IN or OUT] set of the assignment statement on line 8?

Guard / Symbolic Value Information

What guards exist between all of the definitions of the variable X and the writeln on line 25?
Is the writeln on 25 explicitly guarded against the case in which $X \leq 0$?

Figure 18: Examples of questions that CHIRON can answer.
How CHIRON Answers Some of the Questions Raised by PROUST's Evaluation

PROUST's evaluation raised the following problematic areas, each of which is addressed in a planned or prototyped improvement of CHIRON over PROUST.

**Interface:**
What type of interface would make an automatic debugger a better aid to debugging and a better teacher of the skills of debugging than PROUST?

**Knowledge base:**
How can we incorporate lexical, data, and control flow knowledge into a debugger's knowledge base of programming plans and goals?

**Bug description:**
How should bugs be represented so that the debugger's knowledge of errors is more general, and specific "tweaks" are less necessary?

**Control structure:**
What type of control structure would allow a debugger's interpretation process to be more flexible than PROUST's, and allow it to recover gracefully from a failure to match a first goal?

CHIRON's improvements over PROUST can be summarized as follows. First, CHIRON's interface allows the student to ask questions. Second, the layering and hierarchy of CHIRON's knowledge base both allowed us to include lexical, data, and control flow knowledge, and to perform bug identification without a bug catalog. CHIRON focuses on finding the correct parts of the student program, rather than the incorrect; bugs are simply the parts or attributes of the student program that are not correct. Finally, CHIRON's more robust control structure owes its strength to the use of machine learning techniques for plan identification.

The following sections describe CHIRON's improvements in each of these four areas.

**CHIRON's Interface Allows the Student to Ask Questions**

As a first step toward constructing an interface better than PROUST's we have implemented an interactive, question-answering, interface for CHIRON. CHIRON's simple natural language interface parses English questions typed by the students using a semantic grammar (Brown, Burton, and de Kleer, 1982). Figure 18 shows the types of questions that CHIRON is designed to answer. We believe that a question-answering interface will be more effective than PROUST's output which consists merely of a list of the errors found in the program. The PROUST interface (see Figure 3) provides no interaction with the student about the program analyzed.

Letovsky (1986) studied verbal protocols gathered from professional programmers engaged in the task of understanding correct programs. He observed a small set of recurrent behaviors that included asking certain questions of the code and documentation and making certain types of conjectures about the code. We are trying to develop CHIRON's interface so that it supports the type of question-asking/conjectureFORMATION that Letovsky saw, and we are trying to expand on Letovsky's question types so that questions about incorrect code can also be supported.
At the very least, an interactive interface will be useful to us, the future evaluators of CHIRON, simply because an interactive question-answering interface will provide us with an automatic method for collecting more information about students; student transcripts will show the types of questions the students asked CHIRON.

**Hierarchy and Layering in the Knowledge Base**

CHIRON’s plan knowledge is arranged in hierarchical, tree structures. Plans are defined as specializations of more general plans. For example, finding the maximum of a sequence of values is a specialization of a more general plan called linear_search. Figure 19 shows several CHIRON plans, with downward lines running from general plans to their specializations. The most general accumulator plan accepts a sequence of values from another part of the program (called an enumerator), combines the sequence together using some operation, and then returns a single result.

![Diagram of CHIRON's knowledge base]

**Figure 19:** Plan definitions in CHIRON's knowledge-base with generalization/specialization relationships.

An accumulator plan has the parameters listed in Figure 20. These parameters must obey the constraints shown in Figure 21. The definition of the accumulator plan is then simply a function which associates the parameters with the constraints.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?Var</td>
<td>variable into which the result of the accumulation is collected.</td>
</tr>
<tr>
<td>?InitExpr</td>
<td>an expression describing the value to which ?Var is initialized.</td>
</tr>
<tr>
<td>?Init</td>
<td>the part of the program in which ?Var is given the result of ?InitExpr.</td>
</tr>
<tr>
<td>?UpdateExpr</td>
<td>the part of the program describing how ?Var is updated.</td>
</tr>
<tr>
<td>?Update</td>
<td>part of the program in which ?Var is given the result of ?UpdateExpr.</td>
</tr>
<tr>
<td>?EnumVar</td>
<td>successively contains each of the sequence of values that are to be accumulated.</td>
</tr>
<tr>
<td>?EnumInit</td>
<td>part of the program in which ?EnumVar is initialized.</td>
</tr>
<tr>
<td>?EnumUpdate</td>
<td>part of the program in which ?EnumVar is updated to its next value.</td>
</tr>
<tr>
<td>?GuardTest</td>
<td>a boolean expression, true when ?Update can legally be executed, false when it cannot</td>
</tr>
<tr>
<td>init</td>
<td>constrains ?Init to be a type of statement in which ?Var is given ?InitExpr's value.</td>
</tr>
<tr>
<td>update</td>
<td>constrains ?Update to be a type of statement in which ?Var is given ?UpdateExpr's value.</td>
</tr>
</tbody>
</table>

**Figure 20:** Parameters of the accumulator plan
Once the accumulator plan is defined, plans for specializations are relatively simple. For example, a counter is a special type of accumulator in which the ?UpdateExpr increments the value of ?Var. A guarded_counter is a special type of counter in which the count ?Var is updated if and only if ?GuardTest is true. A maximum is a special type of linear_search in which the ?Var is updated if and only if ?EnumVar's current value is greater than ?Var's current value. A linear_search is a special type of accumulator in which the ?Var is updated by replacing its previous value with ?EnumVar's current value, and in which ?Var is updated if and only if ?GuardTest is true.

Hierarchically Arranged Non-Plan Knowledge. Some of CHIRON's non-plan knowledge is also arranged hierarchically. For example, <+, <-, <*, and </> are specializations of the <arithmetic_operator>. Another general pattern <boolean> has specific operators like = and <>. Language specific statements, like PASCAL's <READ>, <READLN>, and <:=> statements are all specific ways of giving a variable a value and so they are all grouped under a language-independent pattern called <variable_gets_value!>. Lexical knowledge can also be included in hierarchies of the same format. For example, variable names like MAX, MAXIMUM, MAXM could all be grouped under the general pattern <variable_name_for_maximum>, which in turn could be declared to be a specialization of <variable_name>. Although, our current CHIRON does not contain lexical knowledge, generalization/specialization hierarchies of this sort are the manner in which we intend to include lexical knowledge.

Layered Plans. PROUST's plans specify the following three layers (similar to that used in the Programmer's Apprentice (c.f., Rich, 1981)).

1. Template Layer: patterns that can directly match code in the students' programs (e.g., ?update part of the counter plan).

2. Flow Layer: descriptions of data and control flow between the pieces of the template layer, for example, that data must flow from the :init part to the :update part (e.g., init->update part of the accumulator plan).
3. Input/Output Layer: information about the type of data produced and consumed by the parts of the template layer (e.g., enum_update->update step of the accumulator plan).

**Bug Description = Correct Description + Generalizations**

PROUST's objective is to determine what is *wrong* with a student's program by explaining discrepancies using bug-rules: knowledge about typical programming errors. In contrast, CHIRON's objective is to explain what is *right* about a student's program by explaining discrepancies using generalization mechanisms: knowledge about the elaboration and simplification of typical programming plans and plan components. CHIRON's use of generalization knowledge to describe bugs allows CHIRON to locate buggy pieces of code in a student program without using bug rules that might be too idiosyncratic or weak (as sometimes is the case with PROUST's bug rules).

From a PROUST perspective there are two problems with the counter plan implemented in the buggy program shown in Figure 16:

\[
\text{COUNT} := \text{COUNT} - 1
\]

was found instead of the expected

\[
\text{COUNT} := \text{COUNT} + 1
\]

and, instead of just one statement, there are two matching the counter update pattern.

In contrast, when CHIRON determines that the \( <+> \) operator does not exist in the \( \text{COUNT} := \text{COUNT} - 1 \) statements, it replaces its expectation with a more general operator (here \(<\text{additive\_operator}>) stored in the operator hierarchy. The buggy lines 11 and 12 of Figure 16 are then matched by CHIRON's expectation. The answer to Question 2 in Figure 17 shows how CHIRON uses this generalization in explaining the bug.

If lines 11 and 12 had been of the form \( \text{COUNT} := \text{COUNT} \times 1 \), then even the more general \(<\text{arithmetic\_operator}>\) would be used. For a *Read for Increment* bug (writing \text{Read(COUNT)} instead of \( \text{COUNT} := \text{COUNT} + 1 \)), CHIRON would replace \( \langle := \rangle \) with the generalization \(<\text{variable\_gets\_value}!>\).

CHIRON could also use its hierarchies to find buggy tests in which, for example, the student had written \( \text{TOTAL} \geq 0 \) instead of \( \text{TOTAL} > 0 \), while PROUST needs a special bug rule: the Sloppy Guard bug rule shown in Figure 9.

In this short introduction to CHIRON's bug description method we have shown how just a small piece of generalization/specialization knowledge about operators could be used to replace four different bug rules that PROUST would have to have in order to find the same errors in students' programs. We are incorporating generalization/specialization knowledge about, not only operators, but plans, data and control flow relations, variable names, and various other objects and relations. An extensive knowledge about generalizations and specializations will allow CHIRON to find errors in student programs without a pre-enumerated catalog of bugs like PROUST has.

---

**A Control Structure that can Never Stall**

Roughly, CHIRON's bug identification control structure consists of the following algorithm:
repeat
  if no unmatched student-program parts
    then
      exit, returning matches of student-program parts to exact and generalized specification parts
  else
    get an unmatched specification part;
    attempt to match to a student-program part.
    If match fails
      then
        if generalization of specification-part exists
          then
            generalize that part
          else
            match student-program part to <anything>
    Because, the symbol <anything> is predefined to be the generalization of everything in the knowledge base, it will match anything, and this control structure can never stall. Pragmatically, CHIRON would still be able to tell the student that an error had occurred in the particular part of the program matched to <anything>.

CONCLUSIONS
Before evaluating PROUST we knew that we wanted our next debugger to be better than PROUST. But, it is only now, after the evaluation, that we have some inkling of what a "better" debugger is. This evaluation has focused our attention onto a concrete set of PROUST's strengths and weaknesses. We started our testing of PROUST with two general questions: (1) Can PROUST explain to students the errors it has found, and (2) Can PROUST identify bugs in novices' programs? We sharpened both of these questions using micro-analyses to uncover why PROUST was able to explain and identify errors in students' programs. Our newest debugger, CHIRON, is an effort to synthesize our evaluation results into a "better" debugger.

Bug Explanation
We found that students who had access to PROUST when they were doing their programming assignments performed modestly better on a midterm examination specifically designed to exercise bug identification and repair skills. At a finer level of interpretation, we saw that PROUST helped students identify idiosyncratic (but not common) bugs in their own programs. And, we found that students who succeeded in fixing particular bugs on their programming assignments were significantly better at repairing similar bugs on the midterm.

The particularity of these results showed us the utility of making hair splitting distinctions between different kinds of bugs: even bugs that are seemingly only slightly different on the surface can be caused by very different factors, and these causal factors can determine how easy it is for a bug identifier to explain a bug to a student. In order for a bug identifier to more ably reason about different kinds of bugs, it is arguably necessary for it to have a depth of knowledge
that is deeper than PROUST's in at least two dimensions: (1) knowledge about *programming* and, (2) knowledge about the *student*. CHIRON's three-layered plan language is two layers deeper than PROUST's plan language and represents a push to put more knowledge about programming into our new debugger. Studies by Spohrer(1989) have given us new insights into novice programmers' reasoning and problem solving strategies. However, it is still not exactly clear how this deeper knowledge about students can be used to increase a bug identifier's effectiveness at diagnosing and explaining bugs to students.

Letovsky (1986) showed us that programmers tend to ask questions and formulate and investigate certain types of conjectures about the code they are reading. Consequently, it seems that PROUST's "edict-style" output could be improved upon. CHIRON's interface is being developed to allow the student to pursue a more natural question-and-conjecture-style inquiry about the errors in their programs. Also, in this evaluation, as evaluators, we found that PROUST's output style did not allow us to determine which parts of PROUST's output, if any, students actually read or understood. Hopefully, students' dialog transcripts with CHIRON will give a more accurate trace of what kinds of remedial advice students ask for, understand, and use. We hope that Littman's(1989) studies of tutorial dialogs will shed further light on the issue of how to make CHIRON educationally effective.

**Bug Identification**

Both our evaluation and the evaluation conducted by Johnson(1986) showed that, overall, PROUST was only able to identify about one half or less of the bugs present in the student programs submitted for analysis. We conducted a number of micro-analyses to cast more light on PROUST's overall accuracy rating: to pinpoint when and why PROUST was (in)accurate in identifying bugs. We found that PROUST was very accurate when it was able to give a full interpretation to a student program and very inaccurate when it was only able to give a partial or incomplete interpretation.

During our evaluation and Johnson's evaluation various small changes were made to PROUST, like strengthening key bug rules. Both we and Johnson found that these "tweaks" to PROUST could make it much more accurate in identifying errors: about 20% better. We are trying to build CHIRON so that it doesn't need "tweaking" by ridding it of PROUST's weaknesses and temperamental parts. For example, CHIRON has a control structure that can't get stalled on the first goal the way PROUST often is; and, in CHIRON we have devised a way to find buggy code in a student's program without using bug rules.

We performed several speculative analyses to determine how our new debugger might benefit from a richer knowledge of the lexical content of program variable names, control flow, and data flow analyses. None of this kind of information is used by PROUST in its analysis of student programs, but CHIRON will because we estimate that using this knowledge will make the new debugger more accurate at identifying bugs than PROUST. The extra layers in CHIRON's plan representation language will allow CHIRON to use this information about student programs.
REFERENCES


Johnson, W. L. (1986). PROUST: Intention-Based Diagnosis of Errors, Los Altos, CA: Morgan Kaufmann. <is that correct spelling of "Intention"?>


Spohrer, J., & Soloway, E. (1986). Novice mistakes: are the folk wisdosms correct?, *Communications of the ACM*, 29 (7), 624-632.


NOTES

Acknowledgements

Our evaluation of PROUST was done at the Yale University Computer Science Department within Elliot Soloway's Cognition and Programming Project (CAPP). We owe our thanks to many former members of CAPP. An earlier technical report (Sack, et al., 1986) contained most of the material that we reported here as our section on the educational evaluation of PROUST. We would like to thank David Littman and Jim Spohrer for their guidance in shaping the form and content of the earlier technical report and, by precedent, the form and content of that section of this paper. Lucian Hughes, Scott Fertig, and Andrew Lilies were also co-authors of the earlier technical report and contributed several months of their time to gathering statistics and critiquing earlier drafts of this paper written in 1986. In particular the data on variable names was Lucian's work. Lewis Johnson lent his expertise to our evaluation efforts when we started in 1985. Stan Letovsky and Rob Rist gave us some pivotal ideas in several 1985 meetings on PROUST. We thank Drs. Eva Baker and Nancy Atwood of the UCLA Center for the Study of Evaluation: they provided critically important moral and technical support for this evaluation effort. Thanks to Prof. Harald Wertz for his advice and support during Warren's year at the University of Paris 8. Drs. Jill Larkin, Carol Scheffic, and Ruth Chabay rescued us from our gaffes and steered us towards coherence with their expert editing advice and enormous amount of encouragement.

Current Authors addresses:

Sack:  Current address: Universite Paris VIII, Departement d'Informatique, 2 Rue de la Liberte, 93526 Saint Denis, Cedex 02, France.
Soloway:  Current address: University of Michigan, Department of Electrical Engineering and Computer Science, Electrical Engineering and Computer Science Building, Ann Arbor, Michigan 48109-2122.