Understanding and Debugging Novice Programs

W. Lewis Johnson
USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292, USA

ABSTRACT

Accurate identification and explication of program bugs requires an understanding of the programmer's intentions. This paper describes a system called PROUST which performs intention-based diagnosis of errors in novice PASCAL programs. The technique used involves generating possible goal decompositions for the program, matching them against the program, and then proposing bugs and misconceptions to explain the mismatches. Empirical studies of PROUST's performance show that it achieves high performance in finding bugs in nontrivial student programs.

1. Introduction

Learning to program is a time-consuming and frustrating process for most novice programmers. One reason for this is that they have to expend so much effort in debugging their programs. Program bugs hinder the learning process in two ways. First, the students are distracted from the curriculum material that they are trying to understand when the programs that they write have bugs unrelated to the concepts being learned. Second, novices frequently have misconceptions about programming language syntax and semantics, which lead to confusions when their programs behave differently from what they expect. It is extremely difficult for novices to discover on their own the misconceptions which account for the unexpected behavior.

Bugs need not be a hindrance to novice programmers, however. If a tutor were to supervise the students' work and provide assistance when the students make mistakes, then errors might even enhance the learning process. Performance errors provide a unique opportunity for the teacher to understand the students' confusions and misconceptions [4]. Given such an understanding, the teacher can then focus on remedying the student's problems, clearing the way for further progress through the curriculum.

Unfortunately, it is rarely possible to provide each student in a programming course with an individual tutor. What is needed instead is a computer program

Artificial Intelligence **42** (1990) 51–97
0004-3702/90/$3.50

which can serve in the tutor's role. Such a program would analyze the students' programs, looking for bugs and bad programming style. It would then help the students overcome the misconceptions that were responsible for the incorrect code.

The process of analyzing programs for syntactic errors is well understood at this point; techniques exist which do a fairly good job of identifying syntactic errors [2, 13] and of correcting them [6, 15]. Semantic and logic errors, on the other hand, are not so easily diagnosed. Most semantic and logical error detectors focus on narrow ranges of bugs, such as uninitialized variables [10] or spelling errors [32]. These errors all share the property that one can detect them regardless of what the intended functionality of the program is. However, many logical errors result in programs which function, but which fail to compute the desired results.

This paper will argue that in order to reliably diagnose as near to the complete range of semantic and logical errors as possible, a debugging system must understand the programmer's intentions. A program is a designed artifact; as such, its design must be taken into account when analyzing it for bugs. The program has an intended function, and has been constructed in order to achieve this function. Debugging should focus on whether the intended function and design of the program are correct, and whether these intentions have been properly executed. Such an approach makes it possible to detect more bugs, and to explain better how to correct them. By relating bugs to the student's intended design, it may be possible to help students learn to design their programs better.

This paper will show that intention-based analysis can be an effective method for diagnosing bugs in programs. It requires knowledge of how to write programs, what errors novice programmers are likely to make, and some general understanding of what a given program is supposed to do. Given such knowledge, one can identify the intended function of each statement in a program, what bugs are present, and where they are manifested. A system called PROUST will be described which determines the intentions underlying novice programs and uses this understanding of intentions to perform accurate analyses of nonsyntactic bugs. The results of empirical evaluations of PROUST on student programs will be presented; these results will demonstrate the effectiveness of the approach.

1.1. Intention-based analysis of an example buggy program

To see why understanding programmers' intentions is important in diagnosing bugs, let us examine the bugs in an example novice PASCAL program. The program to be considered here is a solution to the Rainfall Problem that was assigned in an introductory PASCAL course.

Rainfall Problem. Noah needs to keep track of rainfall in the New Haven area

in order to determine when to launch his ark. Write a PASCAL program that will help him do this. The program should 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 data until the user types 99999; this is a sentinel value signaling the end of input. Do not include the 99999 in the calculations. Assume that if the input value is nonnegative, and not equal to 99999, then it is valid input data.

This problem requires that the students write a program which reads in a series of numbers, each of which represents the amount of rainfall on a particular day. Input termination is signaled when the user types the value 99999. The program is supposed to check the input for validity, compute the average and the maximum of the input, and count the total number of valid inputs and the number of positive inputs. The program must prevent the final 99999 from being included in the computations. This problem thus tests the students' ability to combine a variety of computations into a single working program.

Figure 1 shows a solution to the Rainfall Problem written by a novice programmer. We will refer to this example repeatedly throughout this paper when discussing PROUST. This program has a number of different bugs; however, there is one set of bugs that is of particular interest. Instead of there being a single loop which reads the data, processes it, and checks for 99999, there are two. One is a repeat loop, starting at line 8 and ending at line 28. The other is a while loop contained within the repeat loop. The inner while loop is an infinite loop; it tests the variable RAIN against 99999, but never modifies RAIN.

Well-known analysis techniques such as data flow analysis [10] are capable of detecting the infinite loop in the program in Fig. 1. However, if an infinite loop is a manifestation of a more significant programming error, then simply pointing out the infinite loop may distract the student away from correcting the true error. In particular, if a loop does not belong in the program at all, then the question of whether or not the loop is infinite is moot. Our empirical studies of how students debug programs indicate that novice programmers tend to correct the surface manifestations of bugs rather than the bugs themselves; thus proper descriptions of bugs are crucial.

We believe that the proper analysis of the bug in this program is as follows. The student probably did not intend the while statement at line 19 to loop at

```
 1 program Rainfall (input, output);
 2
 3 var
 4   Rain, Days, Totalrain, Raindays, Highrain, Averain: real;
 5
 6 begin
 7   Rain := 0;
 8   repeat
 9     writeln ('Enter rainfall');
10     readln;
11     read (Rain);
12     while Rain < 0 do
13       begin
14         writeln (Rain:0:2, 'is not possible, try again');
15         readln;
16         read (Rain)
17       end;
18
19     while Rain <> 99999 do
20       begin
21         Days := Days + 1;
22         Totalrain := Totalrain + Rain;
23         if Rain > 0 then
24           Raindays := Raindays + 1;
25         if Highrain < Rain then
26           Highrain := Rain
27       end;
28   until Rain = 99999;
29
30   Averain := Totalrain/Days;
31
32   writeln (Days:0:0, 'valid rainfalls were entered');
33   writeln ('The average rainfall was', Averain:0:2, 'inches');
34   writeln ('The highest rainfall was', Highrain:0:2);
35   writeln ('There were', Raindays:0:0, 'in this period');
36 end.
```

Fig. 1. A buggy solution to the Rainfall Problem.

all. Instead, the effect of an if statement was intended. Since the input statements are at the top of the repeat loop, and the loop exit test is at the bottom of the repeat loop, there has to be a way for control to skip from the input statements to the end of the loop when 99999 is read. Otherwise the 99999 would be processed as if it were a rainfall amount. An if statement would serve this purpose, but the student has written a while statement instead. Empirical analyses of novice programming errors [19] have shown that novice programmers commonly confuse the meanings of if and while, particularly when they are embedded within other loops. If this bug were to be described to the novice programmer, the description should focus on the probable misconcep-

tion about the meanings of if and while, rather than just on the fact that the while statement is an infinite loop.

The key difference between the above analysis and the analysis that a data flow analyzer would produce is the following. A data flow analyzer analyzes the behavior of the program itself, independent of the intentions underlying it. In the above analysis, the focus was on understanding the intentions underlying the program, and then on referring to those intentions in diagnosing bugs. Thus the bug analysis presented above is an *intention-based* analysis; a data flow analyzer, in contrast, performs non-intention-based analysis.

Figure 2 shows PROUST's output when analyzing the program in Fig. 1. The output demonstrates PROUST's ability to understand the intentions underlying the program. PROUST finds all of the bugs in the program, including the while-for-if bug. Although PROUST's output indicates the possibility that the

```
NOW BEGINNING BUG REPORT:

>> Now Reporting CRITICAL Bugs in the SETUP part of your program:

You left out the initializations for the variables HIGHRAIN, DRYDAYS, RAINDAYS, and
TOTALRAIN. Programs should not fetch values from uninitialized variables! If you don't initialize
these variables to the values that you want, you will have no way of knowing what values they
will be initialized to.

(TO CONTINUE, PLEASE PRESS CARRIAGE RETURN)

>> Now Reporting MINOR Bugs in the SETUP part of your program:

The initialization at line 7 appears to be unnecessary.
The statement in question is:
    RAIN := 0

(TO CONTINUE, PLEASE PRESS CARRIAGE RETURN)

>> Now Reporting CRITICAL Bugs in the CONTROL part of your program:

You used a WHILE statement at line 19 where you should have used an IF. WHILE and IF are
NOT equivalent in this context; using WHILE in place of IF can result in infinite loops.
The statement in question is:
    WHILE RAIN <> 99999 DO . . .

(TO CONTINUE, PLEASE PRESS CARIAGE RETURN)

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

The maximum and the average are undefined if there is no valid input. But lines 34 and 33
output them anyway. You should always check whether your code will work when there is no
input! This is a common cause of bugs.

You need a test to check that at least one valid data point has been input before line 30 is
executed. The average will bomb when there is no input.

BUG REPORT NOW COMPLETE.
```

Fig. 2. PROUST's output for the program in Fig. 1.

program will enter an infinite loop, it does not analyze the program by looking for infinite loops. Rather, it tries to understand the intended role of each component of the program, and in the process discovers a while statement that appears to have the intended function of an if statement. Once the bug is found, PROUST can then proceed to explain how the bug will be manifested in incorrect program behavior.

1.2. The principal components of intention-based diagnosis

We will now look at what a system needs in order to be able to perform an intention-based analysis such as the one that we have just seen. The particular mechanisms which PROUST uses to perform the analysis will be introduced. Further discussion of these mechanisms will appear later in the paper.

1.2.1. *Problem descriptions*

One of the things which an intention-based analysis system must do, as indicated above, is to determine what the intended function of the program is. It is difficult to infer the intended function of a program just by inspecting the program; there is no way of knowing whether the program's behavior is really what the programmer had in mind. One needs some way of forming expectations about what the program functionality ought to be. In PROUST the expectations are provided in the form of a description of the problem that was assigned to the students. It is assumed that the students' intended functionality will be reasonably close to what was stated in the problem.

Problem descriptions, for PROUST, are sets of goals to be satisfied, and sets of descriptions of the data objects that test goals apply to. Figure 3 shows one of the problem descriptions that PROUST uses, the description of the Rainfall Problem.[1] These problem descriptions define data objects which the program will manipulate, and some goals to be achieved on those objects. For example, *Output*(*Average*(?DailyRain)) specifies that the average of the rainfall inputs should be computed and output, where the rainfall input is referred to as the

```
?DailyRain isa Scalar Measurement.

Achieve the following goals:
  Sentinel-Controlled Input Sequence(?DailyRain, 99999);
  Input Validation(?DailyRain, ?DailyRain < 0);
  Output(Average(?DailyRain));
  Output(Count(?DailyRain));
  Output(Guarded Count(?DailyRain, ?DailyRain > 0));
  Output(Maximum(?DailyRain));
```

Fig. 3. The Rainfall Problem in PROUST's problem description notation.

[1] The syntax of the description has been altered to make it more readable.

object ?DailyRain. Note that goals implied by the listed goals, such as checking for division by zero when the average is computed, are omitted. Explicitly mentioned goals are more likely to match the students' intentions than implied goals, which the students often overlook or get wrong.

1.2.2. *Hypothesizing goal decompositions*

Given a problem description, the task of identifying the intentions underlying a program amounts to answering the following questions:

- How do the goals in the problem description relate to the goals that are actually implemented in the program?
- How did the programmer intend to implement these goals?

That is, general expectations about the intended function of a program must be refined into a specific account of the functionality and design of the program.

Although the problem description helps determine what the intended function of the program is, it says nothing about how that function is to be implemented. In fact there is nothing it could say, because each student is likely to implement the problem goals in a different way. In small programs it may be possible to enumerate the different ways of solving the problem, but in more complex problems such as the Rainfall Problem the number of possible solutions is too great. When an intention-based diagnosis system works in a complex domain such as PROUST's, it cannot rely solely on a canned description of possible solutions. Instead, it must be able to construct a description of the intentions underlying each individual student solution.

In order to construct descriptions of novice intentions, PROUST relies upon a knowledge base of programming plans. Programming plans, as defined by Soloway, are stereotypic methods for satisfying programming goals [29]. Rich's programming cliches serve a similar function [22]. PROUST's plan knowledge base was constructed as a result of studying commonly occurring patterns of code in PASCAL programs, and from examples culled from programming textbooks. PROUST combines these plans into possible implementations for each goal, and then matches the plans against the program. If the student's code matches one of the predicted plans, then PROUST concludes that the student's intended implementation matches fairly closely to the plan that matched.

When PROUST combines plans into predictions of how the student implemented the problem goals, it is said to be generating possible *goal decompositions* for the problem. A goal decomposition relates the goals that a program is supposed to achieve to the plans that achieve it. In the process of going from goals to plans, it may be necessary to break goals into sets of subgoals, combine related goals into a larger goal, and add goals that are not explicitly stated in the problem. For nontrivial problems, there is often a large number of possible goal decompositions.

```
write('Enter rainfall value:');
read(Rain);
while Rain <> 99999 do
  begin
    if Rain < 0 then
      writeln('Invalid input, try again');
    if Rain > 0 then
      Raindays := Raindays + 1;
    if Highrain < Rain then
      Highrain := Rain;
    if Rain >= 0 then
      begin
        Totalrain := Totalrain + Rain;
        Days := Days + 1;
      end;
    write('Enter rainfall value:');
    read(Rain);
  end
```

Fig. 4. An alternative way of combining input and input validation.

An example of where goals can be combined in different ways in the Rainfall Problem is in deciding whether the goal of inputting the rainfall data and the goal of checking it for validity should be combined. If the two goals are combined into a single plan, then a program such as the one in Fig. 1 results. There the contiguous block of code from line 10 to line 17 reads, tests, and then re-reads the data. If the *Input* goals and the *Input Validation* goal are not combined, then they may wind up in separate parts of the program, as in the example in Fig. 4.

It should be emphasized that the goal decomposition that PROUST hypothesizes for a program need not *correctly* implement the goals in the problem description. The student may have decomposed goals improperly, or have used an inappropriate plan. In such cases PROUST's goal decomposition should still reflect what the student did. PROUST's programming knowledge base is therefore extended so that it can generate incorrect goal decompositions. PROUST is thus able to predict some kinds of bugs as it constructs goal decompositions. Not all bugs are recognized in this fashion, but a significant number are.

1.2.3. *When predicted intentions fail to match*

Even though PROUST generates a number of goal decompositions for each goal, there is no guarantee that any of them will match the student's program exactly. In fact, mismatches are what most often provide clues that there are bugs in the program. If we have chosen the right goal decomposition, and it fails to match the program, then the mismatches can be explained as failed attempts on the part of the programmer to implement the goal decomposition in the code.

SENTINEL READ-PROCESS REPEAT PLAN

Constants: ?Stop
Variables: ?New
Template:
repeat
subgoal Input(?New)
subgoal Sentinel Guard(?New, ?Stop, ?*)
until ?New = ?Stop

Fig. 5. A plan for implementing *Sentinel-Controlled Input Sequence*.

PROUST detects the possibility of a while-for-if bug in the example program by matching different goal decompositions against the program. The goal that PROUST tries to decompose is the *Sentinel-Controlled Input Sequence* goal, the goal of reading in a sequence of numbers until some designated sentinel value is reached. It constructs several goal decompositions for this goal, some using while loops, some using repeat loops; it also tries different ways of structuring the loop. The closest decomposition that it finds uses the SENTINEL READ-PROCESS REPEAT PLAN, shown in Fig. 5. PROUST first matches the repeat statement pattern in the plan against repeat statement at line 8 in the program. It then selects plans to implement the subgoals in the plan, *Input* and *Sentinel Guard*. No plan for implementing the *Sentinel Guard* subgoal matches the program. All of PROUST's plans for implementing *Sentinel Guard* require that there be an if statement to test for the sentinel value; no such if statement appears in the program.

Now, in order to make sure that the student's program is properly understood, some knowledge of common student errors is needed. We need to be able to recognize that the while loop in program could be a buggy implementation of the expected subgoal. In PROUST this knowledge is represented as a knowledge base of production rules, called *plan-difference rules*. These plan-difference rules are responsible for suggesting bugs and misconceptions which account for the mismatches. One such rule, a rule for recognizing when while statements were used in place of if statements, is paraphrased in Fig. 6. Plan-difference rules either account for the differences between the plan and the code by means of bugs and misconceptions, or suggest a way to transform the plan to make it fit the programmer's apparent intentions better.

IF a while statement is found in place of an if statement,
AND the while statement appears inside of another loop,
THEN the bug is a while-for-if bug, probably caused by
a confusion about the control flow of embedded loops.

Fig. 6. Paraphrase of a plan-difference rule for explaining while-for-if bugs.

1.3. Summary of PROUST's approach

To summarize, intention-based errors diagnosis, as it is realized in PROUST, involves performing the following steps:

- generating hypotheses about the intentions underlying the program,
- matching these hypothesis against the code,
- explaining the mismatches.

PROUST is unique in that it can generate a range of hypotheses to test against each program, and because it uses knowledge of common bugs and misconceptions to explain mismatches.

Subsequent sections will explore the different stages of PROUST's analysis in further detail. Section 3 describes the process of constructing goal decompositions. Section 4 describes plan-difference analysis. Section 5 describes how PROUST chooses among alternative interpretations of the program. A more detailed description of each of these processes can be found in [16].

2. Comparing PROUST's Approach to Other Approaches

A number of systems have been built to analyze program errors. Virtually all of these systems are non-intention-based. Instead of identifying the programmer's intentions, they analyze the structure or behavior of the program, and then infer bugs directly from this analysis. In this section some of these other approaches will be compared against PROUST's. In general, other systems cannot recognize as wide a range of bugs, nor can they diagnose bugs as accurately. We will then look at the few systems which are capable of intention-based analysis in other domains, or in other contexts, in order to see how these systems compare with PROUST.

2.1. Non-intention-based approaches

The most common approach to finding nonsyntactic program bugs is to look for anomalous program behavior or structure. The focus here is on programs which can clearly be seen to have bugs, regardless of what the programmer's intentions were. Some systems look for anomalous data flow [10], computations that may not terminate [33], or compare the code against a catalog of common novice mistakes [30]. Others try to interpret runtime errors [14, 32]. Still others analyze program traces for surprising behavior [31]. These systems may be effective for finding certain classes of bugs, but they will not work when the program has no obvious anomalies. Furthermore, they are not very good at pinpointing where the error occurred and why. We saw this in the while-for-if bug in Fig. 1. Without any knowledge of the intended function of the faulty loop, there is no way of knowing whether the exit test of the loop is wrong, whether the body is wrong, or whether a loop was intended at all. Thus a

system which looks for common anomalies will not be able to help a novice programmer realize his intentions in the code.

Another way to find bugs without knowledge of the programmer's intentions is to have the programmer say what is wrong with the program, and have the system try to trace the cause of the bug. The user describes the error by supplying test data which causes the program to generate incorrect output, and indicating the discrepancies between the desired output and the actual output. This approach is used in Eisenstadt's PROLOG Trace Package [8], and in troubleshooting systems such as FALOSY [25], E. Shapiro's debugger [27], and D. Shapiro's SNIFFER system [26]. These systems all assume that the programmer is competent enough to spot any and all incorrect behavior. This assumption is not valid for novice programmers; in fact part of what novice programmers must learn is how to test their programs systematically. A debugging system for novices should be smart enough to find bugs without depending upon the user for assistance.

2.2. Intention-based approaches

In comparison to the number of non-intention-based error diagnosis systems, the number of intention-based ones are few. Those that exist are relatively limited either in their ability to hypothesize intentions underlying programs, or in their ability to handle a wide range of programming errors.

A first step toward intention-based diagnosis is to analyze programs by comparing them against one or more ideal solutions supplied by the instructor. LAURA was an early example of the use of this approach [1]. It was given a single ideal solution for each problem, and compared student solutions against the ideal. Such an approach is acceptable if there is little variability in correct problem solutions, i.e., if the goal decompositions of solutions are essentially the same. In the programming problems that PROUST analyzes, there is simply too much variability for such a scheme to work.

TALUS [23] is another system that compares ideal solutions against programs; it compares each program against a suite of known correct algorithms. TALUS reports the differences between the student's program and the most closely matching correct algorithm. TALUS is similar to PROUST in that it analyzes programs by comparing them against hypothetical algorithms. It differs from PROUST in that all possible goal decompositions must be built in ahead of time, and because it has no knowledge about what bugs and misconceptions are likely to occur in novice programs. Thus TALUS is unlikely to perform as well as PROUST on problems where many variations in goal decompositions are possible, and where students are likely to make mistakes which obscure the intended function of the code. There is a number of different goal decompositions for the Rainfall Problem, resulting from decisions about how to check for boundary conditions, and how to combine the various goals stated in the

problem. Misconceptions about the semantics of PASCAL keywords such as while, if, repeat, begin, and end can frequently result in programs with bizarre structure which is hard to relate to any correct program solution.

The LISP tutor [9] can perform intention-based analysis of errors in LISP programs. It has a model of what the student's current goals are, and updates this model whenever the student makes an edit to the program. If the student makes an incorrect edit to the program, the LISP tutor tries to understand why the student made that change, based upon the tutor's model of the student's intentions. It then corrects the student immediately.

The LISP tutor is successful at diagnosing errors, provided that it understands the programmer's intentions properly. Such an understanding is possible only if the tutor knows what goal the programmer is carrying out at each point in the task. In nontrivial programs, this can be difficult, and the LISP tutor therefore requires guidance from the student. For example, when a student is writing a recursive program using the LISP tutor, the tutor forces the user to select from among a predefined set of recursion plans. The tutor then supplies the student with a program template, which he or she fills in. Recursion plans which do not belong to the predefined set are disallowed.

The advantage of the LISP tutor's approach is that it provides the user with immediate feedback when errors are encountered. The disadvantage is that it restricts the freedom of the student in designing the program. Although PROUST cannot currently analyze recursive programs, it can analyze iterative ones, and it does not require guidance from the user to do so. PROUST also is designed to analyze programs which achieve multiple goals; it does not presume that these goals will be satisfied in any particular order. The LISP tutor is designed to handle programs which achieve a relatively small number of goals; it must assume a small number in order to be able to predict what the user's goal state might be at any given time.

The MACSYMA advisor [13] is similar to PROUST in that it critiques a novice's use of MACSYMA after the novice has attempted to solve the problem. The problems that it analyzes require fewer steps to solve than PROUST's, however. Furthermore, the MACSYMA advisor makes simplifying assumptions about the student's abilities: it assumes that the students' errors are caused only by factual misconceptions about MACSYMA commands. PROUST makes no similar assumption: it is designed to handle the bugs that novice programmers are actually observed to make, regardless of cause. The difference in assumptions results in differences between PROUST's and the MACSYMA advisor's representations of intentions, as we will see later on.

3. Goal Decompositions

This section describes PROUST's goal decompositions, and explains how they are constructed. These goal decompositions are central to PROUST: PROUST's

ability to analyze student programs successfully depends upon its ability to construct goal decompositions which fit these programs. The goal decompositions constitute a model of the student's intentions, a model which is used when identifying and describing bugs. The discussion in this section will proceed as follows. First, the content and purpose of goal decompositions will be discussed. Then the knowledge used in creating these goal decompositions will be discussed. Then the knowledge used in creating these goal decompositions will be described, together with the process which creates them. Finally, the effectiveness of PROUST at recognizing the goal decompositions underlying programs will be assessed.

3.1. The contents of goal decompositions

A goal decomposition is an account of how the goals in the problem are realized in the program. It relates goals to the means by which the goals are implemented; i.e., it relates goals to subgoals and/or plans. The goal decomposition describes why each goal or subgoal arose as part of the solution, e.g., it was dictated by the problem statement, or it was implied by one of the goals in the problem statement. PROUST's goal decompositions thus indicate, for every statement in the program, what goal that statement serves to implement, and in turn how the implemented goal fits into the overall scheme for solving the problem.

In order to see what goes into PROUST's goal decompositions, let us examine the goal decomposition generated by PROUST for the example program in Fig. 1 in some detail. An analysis will be presented of the implementation of the goal *Sentinel-Controlled Input Sequence*, the goal of inputting a sequence of values until a sentinel value is read, in this program. An excerpt of the program relating to this goal appears for reference in Fig. 7.

```
 8 repeat
 9   writeln ('Enter rainfall');
10   readln;
11   read (Rain);
12   while Rain < 0 do
13     begin
14       writeln (Rain:0:2, 'is not possible, try again');
15       readln;
16       read (Rain)
17     end;
18
19   while rain <> 99999 do
20     begin
 .
 .
 .
28     end;
29 until Rain = 99999;
```

Fig. 7. An excerpt of the program in Fig. 1.

PROUST's goal decomposition for this example refers to several goals and plans, each of which will be defined below. The following goals will be referred to in PROUST's goal decomposition:

- *Sentinel-Controlled Input Sequence*: read data and process it until a sentinel value is input;
- *Input Validation*: ensure that input data is valid;
- *Input*: read a single datum;
- *Sentinel Guard*: guard against a sentinel value accidentally being processed as data.

The following plans will be used:

- SENTINEL READ-PROCESS REPEAT PLAN: a repeat loop, in which an *Input* subgoal, a *Sentinel Guard* subgoal, and a set of computations on the input are found (this plan was shown in Fig. 5);
- VALIDATED PROCESS-READ WHILE INPUT PLAN: an *Input* subgoal, followed by a while loop which tests the input for validity, and re-reads it if necessary;
- SENTINEL SKIP GUARD PLAN: an if statement test for a sentinel value.

Note that here, as in the rest of this article, names of goals appear in italics; names of plans appear in capitals.

PROUST's goal decomposition for the example program is as follows.

> The problem description includes two goals, among others: *Sentinel-Controlled Input Sequence* and *Input Validation*. The *Sentinel-Controlled Input Sequence* goal is implemented using the SENTINEL READ-PROCESS REPEAT PLAN. The *Input* subgoal of the plan is combined with the *Input Validation* goal, and the resulting goal is implemented using a VALIDATED PROCESS-READ WHILE INPUT PLAN. This plan matches lines 10 through 17 in the program. The *Sentinel Guard* subgoal of the SENTINEL READ-PROCESS REPEAT PLAN is implemented using a SENTINEL SKIP GUARD PLAN. However, there is a bug in this plan: a while statement was used instead of an if.

PROUST's account of the code in Fig. 7 maps the goals in the problem statement onto plans, and maps the plans onto the code. Each plan is a program template, often containing subgoals which have to be filled in using other plans. Some plans implement a single goal; others, such as the VALIDATED PROCESS-READ WHILE INPUT PLAN, implement more than one goal. Each plan is mapped onto particular lines in the program. The goal decomposition thus identifies the overall design of the program, and what role each statement in the program plays as part of the design. It makes no claims about how the student went about producing this design, e.g., which goals the student attempted first when solving the problem.

3.2. Knowledge used in constructing goal decompositions

In order to generate goal decompositions for novice programs, PROUST requires an extensive knowledge base describing how novices write programs. This knowledge base contains the results of extensive empirical analyses of programs written by novice programmers [3, 17, 19]. The knowledge base is organized as a network of frames, one frame for each plan and goal in the knowledge base. The knowledge base includes plans that novices frequently use to implement programming goals, and it describes common ways in which novice programmers reformulate goals.

3.2.1. *Goal frames*

Goal frames in PROUST list various properties of goals, the most important being the possible ways of implementing the goal. When PROUST is constructing a goal decomposition incorporating a goal such as *Sentinel-Controlled Input Sequence*, it looks up the goal frame to see what alternative implementations are listed there. Each possible implementation is used to construct an alternative goal decomposition.

Figure 8 shows PROUST's description of the goal *Sentinel-Controlled Input Sequence*. The possible implementations are listed in the Implementations slot of the goal frame. Of the six implementations listed, the first four, SENTINEL PROCESS-READ WHILE, SENTINEL READ-PROCESS WHILE, SENTINEL READ-PROCESS REPEAT, and SENTINEL PROCESS-READ REPEAT, are all plans. One of these, SENTINEL READ-PROCESS REPEAT, was employed in constructing the goal decomposition in the previous section. The last two implementations are knowledge structures called "goal reformulations," which will be described later in this section.

The example in Fig. 8 also shows some of the other slots that goal frames typically have. The Form and Main Variable slots define the parameters of the goal: the Form lists all parameters that the goal can take, and the Main Variable

Sentinel-Controlled Input Sequence

Instance Of:	*Read & Process*
Form:	*Sentinel-Controlled Input Sequence*(?New, ?Stop)
Main Variable:	?New
Name Phrase:	"sentinel-controlled loop"
Outer Control Goal:	T
Implementations:	SENTINEL PROCESS-READ WHILE PLAN SENTINEL READ-PROCESS WHILE PLAN SENTINEL READ-PROCESS REPEAT PLAN SENTINEL PROCESS-READ REPEAT PLAN BOGUS YES-NO LOOP BOGUS COUNTER-CONTROLLED LOOP

Fig. 8. A goal.

slot indicates which parameter is the principal input or output of the goal. The Instance Of slot relates goals to more abstract goal classes that they belong to. *Sentinel-Controlled Input Sequence* belongs to the goal class *Read & Process*, which consists of those goals which perform some sort of iterative reading and processing of data. The Name Phrase slot indicates how to describe the goal to a student, in English, should there be a bug in the implementation of this goal. The Outer Control Goal slot gives an estimate of how much code is required to implement the goal; a T here indicates that the code will be one of the larger constructions in the student's program. These estimates help PROUST decide which goals to analyze first in a student's program, through a process described in detail in Section 5.

3.2.2. *Plans*

Just as there is a frame in the knowledge base for each goal, there is a frame for each plan. Currently PROUST's knowledge base comprises more than fifty plans. Each frame contains a plan template, which is a pattern of statements to match against the student's code. As we saw in the SENTINEL READ-PROCESS PLAN in Fig. 5, these templates can have subgoals embedded in them. The subgoals are added to the goal decomposition, after which PROUST generates possible goal decompositions for them in turn, which are then matched against the code. Plans thus serve a dual role in PROUST: they indicate the textual structure that the code must have, and they also indicate the supergoal-subgoal structure of the code.

Figure 9 shows the SENTINEL READ-PROCESS REPEAT PLAN, in greater detail than the version that appeared in Fig. 5. This plan frame has three slots: Constants, Variables, and Template. The Template slot contains the plan template to be matched against the program. This template consists of a repeat statement of the form repeat... until ?New = ?Stop. Contained within the repeat statement are two subgoals, *Input* and *Sentinel Guard*. Their position in the plan template dictates where the code implementing these goals should appear in the program. Since the *Input* subgoal is at the top of the body of the repeat loop pattern, the code that implements the subgoal should be at the top of the

SENTINEL READ-PROCESS REPEAT PLAN

Constants: ?Stop
Variables: ?New
Template:
Mainloop: repeat
Next: *subgoal Input*(?New)
Internalguard: *subgoal Sentinel Guard*(?New, ?Stop, Process: ?*)
until ?New = ?Stop

Fig. 9. The SENTINEL READ-PROCESS REPEAT PLAN, shown in greater detail.

student's repeat loop. The code implementing *Sentinel Guard* should immediately follow the code implementing the *Input* goal, since the *Sentinel Guard* goal immediately follows the *Input* goal in the plan.

All symbols in the plan template preceded by question marks are pattern variables. Pattern variables are bound to data in the student's program when the plan is matched. The Constants and Variables slots are used to declare pattern variables, and to indicate the kinds of data that they match. A pattern variable that is declared in the Constants slot must be bound to a fixed value; for example, the constant ?Stop is bound to 99999 in solutions to the Rainfall Problem. A pattern variable that is declared in the Variables slot must be bound to some varying quantity, e.g., a PASCAL variable. The pattern variable ?New in Fig. 9, which represents the data that the sentinel-controlled loop reads and processes, is declared variable.

As the example in Fig. 9 shows, the statement patterns in plan templates are represented in a form similar to the syntactic structure of the code. For example, the pattern for the repeat statement states specifically that a repeat statement should be matched, and not some other kind of looping statement such as a while statement. This syntactic orientation contrasts with the plan calculus of Rich [22], in which plans are represented in a programming-language-independent form. A syntax-oriented representation is used in PROUST to provide lexical cues for recognizing buggy code in which the syntactic structure is wrong. The while-for-if bug in Fig. 1 is typical here. In the course of analyzing numerous novice PASCAL programs, many programs were encountered where syntactic constructs were either used inappropriately or were misused. Here are some other examples:

- begin-end pairs are sometimes inappropriately used to indicate the boundaries of loops. They appear lexically outside of the loop, rather than inside.
- begin-end pairs are sometimes omitted entirely.
- Extra repeat statements sometimes appear at the end of a program, as if to indicate that control should branch back from that point, as in a BASIC next statement.

In order to interpret programs with bugs such as these, one needs to know exactly which syntactic keywords were used in the program, and where. Thus the relevant syntactic keywords were built into the plan templates. If a syntactic keyword is being used inappropriately, the plan will fail to match. Plan difference rules, as mentioned in Section 1.2.3, can then react to and explain the incorrect keyword usage in the context of the plan. If one were to use a more abstract plan calculus representation for plans, one would then have to maintain two different representations of the same program, one used for plan analysis and another for bug analysis.

Each significant statement or subgoal in a PROUST plan has a label attached to it. In the SENTINEL READ-PROCESS REPEAT PLAN, the repeat statement is labeled Mainloop, the *Input* subgoal is labeled Next, and the *Sentinel Guard* subgoal is labeled Internalguard. These labels are used to characterize the function of each component of the plan. There is a fixed, predefined set of plan labels which are used to annotate all plans in the knowledge base. Mainloop labels, for example, are always associated with the looping statements of plans. Plan labels differ from subgoals in that instead of characterizing the function of each plan component separately, they characterize the role that the component plays within the overall plan. For example, *Input* subgoals can be used either to initialize variables or to obtain successive values for variables. Because the *Input* subgoal in the SENTINEL READ-PROCESS REPEAT PLAN is labeled Next, rather than Init, PROUST can tell that the *Input* subgoal does not initialize ?New, but instead obtains successive values of ?New.

Not all subgoals of plans must be implemented in a specific place in the program, as the *Input* and *Sentinel Guard* subgoals of the SENTINEL READ-PROCESS REPEAT PLAN must be. Some plans leave the location of subgoal implementations unspecified. The AVERAGE PLAN, the ordinary plan for computing the average of a sequence of values, is an example of a plan which does not specify where subgoals should be found. The AVERAGE PLAN appears in Fig. 10. This plan computes the average by dividing the sum of a sequence of values by the count of the number of values in the sequence. The sequence of values is represented by the pattern variable ?New. The goal of computing the sum of a sequence of values is called *Sum*; the goal of counting the number of values is called *Count*. It does not matter where the sum and count are computed, as long as they are computed before the AVERAGE PLAN is invoked. Therefore the *Sum* and the *Count* goals are not specified as plan components; instead, they are listed as "posterior goals," i.e., goals that should be added to

AVERAGE PLAN

Variables: ?Avg, ?Sum, ?Count, ?New
Posterior Goals:
Count(?New, ?Count)
Sum(?New, ?Sum)
Guard Exception(*component* Update *of goal Average*,
((?Count *from goal Count*) = 0))
Exception Condition:
(?Count *from goal Count*) = 0
Template: (*component* Mainloop *of goal Read & Process*)
followed by:
Update: ?Avg := ?Sum / ?Count

Fig. 10. A plan with two different kinds of subgoals.

the goal decomposition, but which are not subcomponents of the plan.[2] The third posterior goal, *Guard Exception*, requires that the average computation not be performed if the count of data items is zero. The Exception Condition slot supplies the conditions under which the results of the plan are undefined, i.e., when the count of data items is zero.

3.2.3. *Goal reformulations*

Suppose that a student fails to follow the problem requirements strictly, and writes a program which reads a fixed number of inputs, rather than reading until 99999 is typed. Such slight deviations in program goals occur with some regularity in programs written by novices. In order to understand such a program, PROUST much recognize that the student's goals deviate from the problem description. Goal reformulations are used to predict and characterize such deviations.

In general, goal reformulations substitute one set of goals for another set of goals. The new goals may or may not be equivalent to the old goals. PROUST's goal for reading and processing a fixed number of inputs is called *Counter-Controlled Input Sequence*, reflecting the fact that a counter-variable is used as the loop control variable in such cases. PROUST reformulates its *Sentinel-Controlled Input Sequence* goal into a *Counter-Controlled Input Sequence* goal in order to construct an accurate goal decomposition for the student's program. Since the student's goal is inappropriate for the problem the goal decomposition must be marked as buggy.

Goal reformulations such as the one going from *Sentinel-Controlled Input Sequence* to *Counter-Controlled Input Sequence* are stored in PROUST's knowledge base as goal reformulation frames. Two of the six implementations for *Sentinel-Controlled Input Sequence*, BOGUS YES-NO LOOP and BOGUS COUNTER-CONTROLLED LOOP, are goal reformulation frames. Figure 11 shows one of

BOGUS COUNTER-CONTROLLED LOOP

Form:
BOGUS COUNTER-CONTROLLED LOOP(?New, ?Stop)
Component Goals:
Counter-Controlled Input Sequence(?Cnt, ?New, ?Max)
Bugs:
Implements Wrong Goal(⟨Lisp code for further describing bug⟩)

Fig. 11. A buggy reformulation of *Sentinel-Controlled Input Sequence*.

[2] The word "posterior" is used because the goals are added to the goal decomposition after the plan template has been matched. This is an efficiency consideration, to ensure that no further development of the goal decomposition is performed until PROUST has determined that the current plan template matches the code.

these, the BOGUS COUNTER-CONTROLLED LOOP reformulation. The reformulation frame contains two principal slots: a Component Goals slot, which lists the goals which will replace the old goals, and a Bugs slot, which describes the bug associated with this goal reformulation. The bug description indicates that the student is implementing the wrong goal; the details of the bug description are provided by a piece of LISP code that is not shown here. Such goal reformulation frames are examples of buggy novice "knowledge" about programming, used to characterize common flaws in novice goal decompositions.

Another example of goal reformulation appeared in the goal decomposition of the program in Fig. 1. There the goals *Input* and *Input Validation* were combined into single goal, implemented using the VALIDATED PROCESS-READ WHILE INPUT PLAN. This is done by first reformulating the two goals into a single goal, called *Validated Input*, and then looking for a plan to implement that goal. To handle cases such as this, reformulation rules are associated with goals such as *Input* and *Input Validation* in the knowledge base. Such rules fire when a goal is being decomposed while other related goals are active, suggesting ways of regrouping the goals.

3.3. Constructing and matching goal decompositions

The discussion will now turn from the knowledge used in goal decompositions to the process of constructing goal decompositions and matching them against the student's code. This process involves searching a state space called the *interpretation space*. The term "interpretation" will be used in this article to refer to the entire body of information gained when analyzing a student's program. The current discussion focuses on two parts of interpretations, goal decompositions and the mapping between goal decompositions and the students' code.

Each state in the interpretation space includes three things:

- an agenda of goals whose implementation in the program has yet to be determined,
- a partial goal decomposition,
- matches between the plans in the partial goal decomposition and the code.

In the initial state, the goal agenda is exactly those goals which are listed in the problem description, and the partial goal decomposition is empty. In the final state, the goal agenda is empty, and the goal decomposition is completely specified. As PROUST traverses the state space, it incrementally elaborates the goal decomposition and matches the plans in the goal decomposition against the program.

The following types of transitions are performed between states:

- goal selection: a goal is selected from the goal agenda;

- goal reformulation: the selected goal, possibly together with other goals, is reformulated;
- plan selection and matching: the selected goal is implemented using a plan, and the plan is matched against the program. Any subgoals in the plan are added onto the goal agenda.

PROUST goes through a cycle of these transitions, alternately selecting goals and either selecting and matching plans or reformulating goals. The set of states that PROUST passes through, together with the transitions between them, constitute a tree, called the *interpretation tree*. The interpretation tree is implemented literally in PROUST as a tree of nodes, each representing a different state in the interpretation space.

The construction of the interpretation tree for the Rainfall Problem begins as follows. PROUST starts by selecting the goal *Sentinel-Controlled Input Sequence*. Then possible goal decompositions involving *Sentinel-Controlled Input Sequence* are identified. From the goal frame for *Sentinel-Controlled Input Sequence* shown in Fig. 8, PROUST finds that there are four possible plans: SENTINEL PROCESS-READ WHILE, SENTINEL PROCESS-READ WHILE, SENTINEL READ-PROCESS REPEAT, and SENTINEL PROCESS-READ REPEAT. There are also two possible buggy goal reformulations: BOGUS YES-NO LOOP and BOGUS COUNTER-CONTROLLED LOOP. PROUST therefore constructs six new interpretation states, four for the plans and two for the reformulations, and links them to the current state. Each state includes an agenda of goals remaining to be processed. In the case of the plan selection states, the goal agenda is the original goal agenda minus *Sentinel-Controlled Input Sequence*. For the goal reformulation states, the agenda also includes the new goals added as a result of the goal reformulation. In the case of the BOGUS COUNTER-CONTROLLED LOOP state, for example, the goal agenda contains the added goal *Counter-Controlled Input Sequence*, which is supposed to be implemented in place of *Sentinel-Controlled Input Sequence*.

The interpretation tree must be expanded until PROUST has identified alternative plans to match against the program. It is through matching the plans that PROUST determines whether or not a particular interpretation fits the student's program. Therefore the goal reformulation states must be expanded, using the same procedure of goal selection, goal reformulation, and plan selection. In the case of the BOGUS COUNTER-CONTROLLED LOOP state, the new goal *Counter-Controlled Input Sequence* is selected. Plans implementing this goal are retrieved. This expansion process continues until every leaf in the interpretation tree is a plan selection state.

Once a set of alternative plans are selected, each plan is matched against the program. Information is added to each interpretation tree node indicating where the plan matched the program, and whether there were any matching errors. At most one match is recorded for each plan. This means that if a plan

matches the program in two places, the node in the tree for that plan must first be copied. Each copy is then annotated to indicate what part of the program it was matched against. When matching the example in Fig. 7, SENTINEL READ-PROCESS REPEAT is one of the plans that is matchable. As was shown in Fig. 9, the matchable part of the plan is the repeat loop; it matches just one statement, the repeat statement at line 8. No node splitting is necessary. In order to complete matching the plan, however, the plan subgoals must be interpreted as well. The interpretation tree expansion therefore continues, this time for each subgoal of the plans that have matched successfully.

Once PROUST has tried each of the alternative plan matches, there will usually be a single plan that matches better than the alternatives. Deciding which match is best is an involved process, which will be described in detail in Section 5. For now let it suffice to say that the goodness of match depends upon how close the match is, whether or not the mismatches can be explained as bugs, and whether the plans' subgoals were interpreted successfully. Then the node describing the best match is selected for further expansion; a new goal is selected, and the process repeats. If PROUST is unable to choose between two alternatives, it will try to back up and select a different goal. In the example in Fig. 9, no backup is necessary; the SENTINEL READ-PROCESS REPEAT PLAN matches better than any of its alternatives. Therefore expansion continues from the point in the tree where the matching of SENTINEL READ-PROCESS REPEAT and its subgoals was completed.

3.4. Goal selection and the ambiguity problem

When PROUST constructs goal decompositions it must be sensitive to problems that the plan matcher may have in matching the plans in the goal decomposition against the code. If a plan is matched without any knowledge of where in the program it should match, there is risk of ambiguous matching. The plan may match more than one section of the program, and PROUST will have no way of knowing which match is the right one. Some simple examples are plans for inputting and outputting data. All input statements and output statements look much alike. PROUST does not attempt to parse variable names or interpret output messages in order to determine what data a given input or output statement is manipulating. Therefore PROUST avoids matching input and output plans until it knows what data the variables in the program refer to.

Match ambiguity results in excessive search of the space of interpretations; since PROUST cannot tell which is the correct match for a plan, it must explore all possibilities. PROUST's plan matcher is designed to take advantage of information gained in interpreting one part of a program when matching plans in other parts of the program. PROUST orders the selection of goals to allow the plan matcher to take maximal advantage of available information about the program, thus minimizing the problem of ambiguity.

3.4.1. *Using context to reduce ambiguity*

Contextual information is supplied to the plan matcher in two ways. First, plans can refer to labeled components of other plans. Second, parameters of goals can be used to bind pattern variables in the plans that implement the goal. The COUNTER PLAN in Fig. 12 can make use of both kinds of contextual information. This plan is the common method for implementing the goal of counting a sequence of values, called *Count*. The template in this plan refers to the component labeled Process of a *Read & Process* goal. The Process component of a *Read & Process* goal is the part of the plan where the data is processed. The COUNTER PLAN states that the counter update must lie inside the Process part of the loop, thus restricting where the plan matcher must look for matches. The COUNTER PLAN has one pattern variable ?Count; this stands for the counter variable. The goal of counting values, *Count*, has two parameters, ?New, which is the quality being counted, and ?Count, which is the resulting count. The parameter bindings in the goal are passed on to the plan. Therefore if PROUST believes that the counter variable is represented by a specific PASCAL variable, say Days, this variable is supplied as a parameter to the goal, e.g., the goal might read *Count*(Rain, Days). The binding of ?Count to days is passed on to the plan, so the statement pattern ?Count := ?Count + 1 becomes Days := Days + 1.

The degree of ambiguity in matching a plan thus depends crucially upon whether or not bindings for plan variables are known beforehand. In the COUNTER PLAN example, the scope of search through the program can be reduced if we know where the *Read & Process* goal is implemented. The counter update can be spotted if we know beforehand what the counter variable is. Otherwise the plan would match any increment statement anywhere in the program.

The ambiguous match of the COUNTER PLAN can be avoided if the plan is matched after the *Sentinel-Controlled Input Sequence* and *Average* goals are analyzed. The *Sentinel-Controlled Input Sequence* goal is a *Read & Process* plan, so once a plan implementing it is matched, the plan matcher can locate the counter update in the body of the loop. The analysis of the *Average* goal provides a binding for the pattern variable ?Count. When PROUST analyzes the average computation, say Averain := Totalrain/Days, it matches it against the

Variables: ?Count
Template:
Init: ?Count := 0
(*in component* Process *of goal Read & Process*)
Update: ?Count := ?Count + 1

Fig. 12. The COUNTER PLAN.

pattern ?Avg := ?Sum/?Count in the AVERAGE PLAN, shown in Fig. 10. Each of the pattern variables in this pattern is bound; in particular, ?Count is bound to Days. The plan indicates that the binding of ?Count should subsequently be used when matching the *Count* goal. Thus when the *Count* goal is analyzed later on, the possibility of ambiguous plan matching is reduced.

3.4.2. *Goal selection strategies to reduce ambiguity*

The success of PROUST in reducing match ambiguity depends upon whether or not it can select goals for analysis in an order so as to take maximal advantage of previous match bindings. It employs the following strategies:

(1) select first those goals which are implemented using large plans,
(2) try to select goals whose parameters are all bound,
(3) try to select goals which do not have potential plan-matching conflicts with other goals on the agenda.

The first of these strategies is responsible for PROUST selecting *Sentinel-Controlled Input Sequence* first for decomposition. The goal frame for *Sentinel-Controlled Input Sequence* indicates that the goal is an outer control goal, i.e., it should match a major control structure in the program, such as a loop. Such major control structures are more likely to appear uniquely in the program, so ambiguity is less of a concern with them. The second and third strategies ensure that the *Count* goal is not selected until the parameters of the goal are all bound. The problem description contains two goals, *Count* and *Guarded-Count*, which are similar to each other, in that they both update counters. Therefore there are potential plan-matching conflicts between these two goals: a plan implementing one goal might match code implementing the other goal. The *Average* goal, however, has no such potential for conflict; plans for implementing the goal are clearly distinct from plans for implementing the other goals. Therefore PROUST favors selection of the *Average* goal over selection of either the *Count* goal or the *CountPositives* goal. Once the *Average* goal has been matched, the parameters of the *Count* goal are all bound to variables in the student's program, so it is now possible to select that goal without worrying about ambiguous matches.

4. Analyzing Plan Differences

If all goes well, the goal decomposition construction process outlined above will succeed in matching a goal decomposition against the student's program. Usually, however, none of PROUST's proposed goal decompositions matches exactly. The match failure can be due to an unexpected bug, or due to an unexpected variant of a plan. The differences between an expected plan and the actual code are called *plan differences*. Most of PROUST's ability to diagnose bugs comes from knowledge of how to explain these plan differences.

In what follows, PROUST's knowledge about plan differences, in the form of plan-difference rules, will be described, together with the mechanism that PROUST uses to invoke plan-difference rules.

4.1. Why plan-difference analysis is needed

Plan differences can arise for the following reasons.

- A goal was improperly implemented, resulting in a bug.
- A goal was implemented in an unusual way, but correctly.

In PROUST, plan-difference rules are used to account for both kinds of plan differences. Some rules simply recognize common plan differences, and describe the bugs or misconceptions that would cause them. Others transform the matched plan into a form that corresponds more closely to the student's implementation, or transform the student's code to make the underlying goal decomposition clearer. PROUST applies plan-difference rules in succession until all plan differences are accounted for. The catalogue of plan-difference rules was constructed as a result of hand analysis of numerous novice PASCAL programs, as described in [19].

We have already seen an example of a plan-difference rule that applies to recognize an improper implementation: the while-for-if bug rule in Fig. 6. Figure 13 shows an example of a correct, but unusual, implementation, which results in a plan difference. In this example, the plans for computing the maximum and the sum have been combined: the sum update is embedded inside the if statement in the maximum computation. This sort of combination of plans occurs fairly frequently in novice programs. When PROUST tries to analyze this code, its plan predictions will fail. PROUST needs to recognize that its plans for computing the maximum and the sum are really applicable here, although they have been combined in an unusual way. PROUST also must recognize that the code is correct, albeit stylistically dubious.

The plan-difference rule in this case is called the Distribution Transformation Rule. It extracts the update statements out of the if statement and combines them into a single statement. The mechanism for doing this is as follows. As described in Section 3.3, PROUST constructs a tree of interpretation of the program. This tree serves as a database context mechanism. PROUST can

```
if Max > New then
  begin
    Max := New;
    Sum := Sum + New
  end else
    Sum := Sum + New;
```

Fig. 13. A variant computation of the sum and maximum of a variable.

```
While (New < 9999) and (New <=0) do
  begin
    Count := Count + 1;
    Sum := Sum + New;
    if 0 < New then
      Rainy := Rainy + 1;
    if New > Max then
      Max := New;
    Writeln('Next value please:');
    Read(New);
  end
```

Fig. 14. Buggy code subject to various possible rule applications.

add assertions about the program to a given node in the tree; these assertions will only be retrievable from that node or one of its descendents. The structure of the program itself is represented as a collection of facts about each statement in the program, facts which can be superceded by subsequent assertions. The Distribution Transformation Rule simply asserts that the two running total statements are below the if statement. It also deletes one of the two statements from the list of statements remaining to be interpreted in the program. Then when the plan is rematched, it matches only one running total update, which now appears in the right position.

Finally, consider a more complex example, the buggy loop in Fig. 14. The while loop here has three bugs:

(1) it uses 9999 rather than 99999 as the sentinel value,
(2) it tries to validate the input and perform an exit test at the same time,
(3) its input validation test is incorrect; it throws out positive data rather than throwing out negative data.

The solution also has a legal variant on the predicted plan: it treats any value greater than or equal to 99999 as a sentinel value. In the predicted plan, 99999 must be typed exactly for the loop to terminate, but the sentinel test in this example is still considered to be satisfactory. PROUST can successfully analyze this code by applying a series of plan-difference rules, one for each observed difference between the predicted plan and the code.

4.2. Plan-difference rules

Plan-difference rules are test-action pairs. The test part matches a plan difference, as well as the context in which the plan difference occurred. The action part accounts for the difference in terms of bugs, plan variants, and misconceptions.

Figure 15 shows one of PROUST's plan-difference rules, the rule for detecting typographical errors in numbers. This rule is trivial in comparison to other

Number Spelling Error Rule

```
Error Patterns:  ((*Const* . *Const*))
Test Code:       (SpellCorrectible (ErrorPattern) (ErrorMatched))
Bug:             (Typo (FoundStmt., (Match-Node (BuggyMatch)))
                       (ExpectedExpr., (ErrorPattern))
                       (FoundExpr., (ErrorMatched))
                       (InterpNode., *InterpNode*)))
```

Fig. 15. A rule for identifying typographical errors.

rules such as the Distribution Transformation Rule; its simplicity makes it that much easier to explain. The rule is represented in a frame-like manner, where some slots describe tests to be performed, and some slots describe actions to be performed. A variety of slots are specifiable as parts of plan-difference rules; this particular rule has only three slots. The Error Patterns slot characterizes the plan difference that the rule applies to. The plan difference is described as a dotted pair, where the car of the pair is what the plan predicted, and the cdr is what was actually found. Here the error pattern refers to a plan difference in which a constant was expected, e.g., 99999, and a different constant was found, e.g., 9999. The Test Code slot is a fragment of LISP code to further test applicability of the rule; this test code fragment checks whether the found constant is plausibly a typographical error. The Bug slot is part of the action part of the rule: it contains a pattern for the bug description to be generated to account for the plan difference. The pattern contains fragments of LISP code which are evaluated at the time when the rule fires; values that are generated become slot fillers for the new bug description.

Rules such as the one in Fig. 15 are so simple because their role is only to acknowledge that the bug is a known bug. The hard work has already been done in the goal decomposition and plan matching. The more complex rules are the ones that transform the code, such as the Distribution Transformation Rule. In these cases, however, the action part of the rule is represented procedurally; a LISP procedure modifies the plan, or makes local assertions about the structure of the program. It would be desirable to implement a more declarative formalism for describing these rule actions.

4.2.1. *Test parts of plan-difference rules*

The plan differences themselves are actually only a small part of what plan-difference rules test for. Altogether, the following kinds of information are used by plan-difference rules:

- plan differences,
- the plan component being matched,
- the plan and goal currently being analyzed, as well as those which have been analyzed previously,

- the structure of the code being matched, or its intended structure,
- bugs and misconceptions which have so far been identified.

Rules can examine the context of the plan in the program, and examine other bugs which have so far been found, in order to select the best interpretation of the plan difference. In general, plan-difference rules need to make use of any information which might shed light on the student's intentions and likely bugs.

The term patterns of plan-difference rules are written primarily in a declarative form. The plan difference, the plan component being matched, and the plan and goal being analyzed all go into the declarative pattern. However, since such a wide range of information must be tested in plan-difference rules, the declarative test-pattern language cannot be used to describe all tests that may be needed. Therefore LISP procedures are also required to perform some of the special-purpose tests, such as the test to see whether a plan difference is a plausible typographical error.

4.2.2. *Action parts of plan-difference rules*

The action parts of plan-difference rules do three kinds of things:

- suggest bugs and misconceptions,
- modify the current plan to make it fit the student's code better,
- indicate how the code should be rephrased to make the student's intentions clearer.

When bugs and misconceptions are suggested, they are added as assertions to PROUST's interpretation tree. PROUST can then see for each goal and plan what bugs and misconceptions arose in their implementation. When it is ready to describe the bugs to the student, PROUST traverses the interpretation tree from the leaf to the root, collecting all of the bug and misconception assertions that it finds there. It then can generate explanations for each bug and misconception to present to the student.

If a rule suggests that the student's plan is a variant on PROUST's plan, the plan-difference rule modifies PROUST's plan to fit the student's code better. For example, consider the test in the maximum computation in Fig. 13:

```
if New > Max then . . .
```

The test which appears in PROUST's MAXIMUM PLAN is slightly different:

```
if ?Max < ?New then . . .
```

The test predicate is reversed from what PROUST expected. A plan-difference rule is required to explain the difference. The rule which applies in this case is

a rule to invert relational expressions, turning expressions of the form ?x < ?y into expressions of the form ?y > ?x. This rule is applied to the plan being matched, the MAXIMUM PLAN, changing the expected test in the plan. The plan is then matched a second time; the second time the plan matches.

Plan-difference rules sometimes suggest ways of rephrasing in the student's code to make the underlying intentions clearer. The while-for-if bug in Fig. 1 requires such a rephrasing. Once PROUST determines that the student intended the effect of an if statement, it must make an assertion in the interpretation tree indicating that the while statement should be interpreted as if it were an if statement. Then the analysis of the remaining code in the loop will not be thrown off by the extra while statement. The same mechanism is used by the Distribution Transformation Rule to handle the merged maximum and running total plans in Fig. 13.

4.3. Application of plan-difference rules

PROUST's plan-difference rule application mechanism must apply those rules which help explain plan differences, and avoid rules which do not contribute to an explanation. It must be able to chain rules and consider alternative rules to apply, but avoid blind application of rules. The example in Fig. 14 will be used to illustrate the kinds of rule application that must be permitted and the kinds that should be avoided. As indicated earlier, the loop exit test, (New < 9999) AND (New <=0), has multiple bugs. Several plan-difference rules must be applied together in order to account for the plan differences. Suppose that one of the while loop plans implementing *Sentinel-Controlled Input Sequence* is being matched, so that a test such as New <> 99999 is expected. First, a rule called the conjoined Loop Bug Rule applies, which suggests that the exit test of the loop is performing two tests at once, which are combined using an AND operator. The actual sentinel-control test should be one of the two clauses of the AND expression. It is not possible for the rule to tell which clause is the desired one, since there are other plan differences involved. Therefore the rule proposes two ways of rephrasing the test in the plan, ((New <> 99999) and ??) and (?? and (New <> 99999)).[3] The plan-difference rule applier then tries matching these new patterns against the code.

When each of the new plan patterns are matched, new plan differences arise. The first pattern fails to match because New < 9999 was found instead of New <> 99999. The second pattern fails to match because New <=0 was found instead of New <> 99999. The second of these matches is a dead end; there is no rule that can relate New <=0 to New <> 99999. Rules can be applied to explain the plan difference of the other pattern, however. A rule called the Sloppy Sentinel Guard Bug Rule applies which suggests that the student may be testing for any value greater than or equal to the sentinel value, rather than

[3] ?? in these patterns is a wild-card pattern variable, matching an arbitrary expression.

testing for the exact sentinel value. The Number Spelling Error Rule explains the difference between the expected 99999 and the actual 9999 as a typographical error. The plan differences are now all accounted for, so the bugs that were found are recorded in the interpretation tree. If any of the plan differences had been left unaccounted for, the entire plan-difference analysis would be scrapped, and PROUST would have to try matching a different plan against the program.

We see in this example that PROUST sometimes has to apply multiple plan-difference rules in succession, yet it cannot be certain that a given rule is applicable until the plan-difference analysis as a whole succeeds. When the Conjoined Loop Bug Rule applies, it can suggest that one of the conjuncts of the and clause is the sentinel test, but it cannot determine which one is the sentinel test. If too many rules suggested too many patterns to try, there would be a multitude of blind alleys in the plan-difference analysis. Such blind application of rules must be avoided if PROUST is to complete its analysis in a reasonable amount of time.

In order to restrict unnecessary rule application, PROUST's rule application mechanism implements the following policies. First, a constraint relaxation approach is used. Each rule is given a ranking. This ranking takes into account whether or not the rule identifies bugs, and the severity of the misconceptions that cause the bugs. It also takes into account whether the rule attempts to explain the plan differences in a statement which has already been matched, or whether it causes the plan matcher to search elsewhere in the program for possible matches. Those rules which do not presume bugs and which apply to existing partial matches are attempted first; then if necessary rules involving progressively more serious bugs and which involve greater amounts of search of the program are considered. Thus the constraints on rule application are progressively lifted until a suitable explanation is found. Similar constraint relaxation schemes have been employed in other debugging systems such as Davis' [7].

The other policy which PROUST's plan-difference rule application mechanism follows is to apply only those rules which appear to help explain the plan differences. PROUST counts the number of syntactic differences between the pattern and the matched statement, and applies only those rules which reduce this number. The rule may still be inappropriate, because the new plan differences are unexplainable; if so, then backtracking will be necessary. Nevertheless, most unnecessary rule applications are avoided by means of this restriction.

Constraining rule application in the manner described above has a price, in terms of missed bug diagnoses. The correct diagnosis is not always the most constrained one, and the correct rule to apply does not always produce an immediate improvement in the plan differences. PROUST may ultimately be forced to apply a wider range of rules in order to achieve greater diagnostic

accuracy. The author has experimented with an alternative approach to rule application, which terminates rule application only when loops appear in the sequence of rules. This approach appears to do a significantly better job of finding valid explanations for plan differences, without substantially increasing the number of unnecessary rule applications.

4.4. Bug descriptions

When a plan-difference rule identifies a bug, it generates a *bug description* characterizing what kind of bug or misconception it is, and the context in which it occurred. These bug descriptions are subsequently used to generate diagnostic explanations for the student. A pattern for a bug description appeared in the plan-difference rule shown in Fig. 15. There are two types of representations which PROUST uses for describing bugs. Superficial representations are used when PROUST is able to locate and characterize the plan difference, but cannot identify the cause simply by looking at the code. Deeper representations are used when PROUST can describe the bug as an error in the student's goal decomposition, or as a misconception.

4.4.1. *Superficial bug descriptions*

Superficial bug descriptions in PROUST use the notations in "Bug collection I" [19] for describing bugs. In this system, the bugs are classified as being either missing plan components, spurious plan components, misplaced plan components, or malformed plan components. The plan components themselves are characterized according to the fucntion that they perform, i.e., initialization, input, output, update, guard, or nonexecutable statement. The function of the plan component comes from the label assigned to that component in the plan, as described in Section 3.2.2.

If PROUST describes a bug using this superficial categorization, it cannot give a very insightful description of the bug to the student. However, the categorization is still useful to PROUST when PROUST compares bugs. For example, when PROUST finds that an initialization is missing, it checks to see whether other initializations are missing. If all initializations are found to be missing, then PROUST can suggest a misconception to the student which explains all of the missing initializations at once. This allows PROUST to generate diagnostic text such as the following:

> You left out the initializations for the variables Highrain, Drydays, Raindays, and Totalrain. Programs should not fetch values from uninitialized variables! If you don't initialize these variables to the values that you want, you will have no way of knowing what values they will be initialized to.

4.4.2. *Deeper bug descriptions*

If a likely explanation for a bug can be found, PROUST describes the bug in terms of that explanation. The following explanatory descriptions of bugs are used:

– Implements Wrong Goal: Some goal was implemented in the program other than the one required by the problem statement. For example, if a student writes a counter-controlled loop instead of a sentinel-controlled loop, this is classified as an Implements Wrong Goal bug.

– Wrong Plan for Goal: The student's plan for implementing a goal is not a recommended one. For example, it may have an unwanted side-effect, such as clobbering a variable which is used later on in the program.

– Misconception: The student's code indicates the presence of a specific misconception. There has to be evidence of a misconception in more than one place in the program for PROUST to classify it as a misconception. This category is further subcategorized according to type of misconception involved. The misconceptions that PROUST can recognize are described in [16]. Examples of the kind of misconceptions that PROUST recognizes are misconceptions about how control flows through while loops, and misconceptions about when variables must be initialized.

– Implements Wrong Goal Argument: A goal is implemented in the student's code, but one of the parameters of the goal is incorrect. For example, an *Input Validation* goal might perform the wrong validity test on the input data.

– Improper Contingent Goal: A goal was implemented contingently, and this resulted in a bug, because one or more cases were overlooked where the goal needed to be implemented. Contingent goal realization is described in [16].

– Wrong Component for Plan: The student's plan for implementing a goal contains a component which is appropriate to a different plan. For example, the student might write a counter-update instead of a running total update.

– Mistransformed Code: The student attempted to transform a plan, e.g., merge a RUNNING-TOTAL PLAN and a MAXIMUM PLAN as in Fig. 13. If the merged code does not work correctly, it is described as mistransformed code.

– Typo: The student made a typographical error.

4.4.3. *Reporting bug descriptions*

After the analysis of a program is complete, the bug descriptions which were created during analysis are collected and reported to the student. First, the bug descriptions are sorted according to the severity of the bug, and the part of the program in which the bug occurs. Then for each bug, an English description is generated. The English is generated using an ordinary phrasal generator, which selects a phrase to generate based upon the type of the bug and the slot fillers of the bug. Blank fields in the phrase are filled in with the names of goals, the line numbers at which statements occur, the names of variables, and other

information which might supply context for the bug. We are also experimenting with other, nontextual methods for describing bugs, e.g., presenting test data which will cause the bug to manifest itself as invalid input-output behavior.

4.5. Factors influencing the form of PROUST's interpretations

We have now examined each component of the interpretations that PROUST constructs for programs. These interpretations contain a variety of information; altogether, they may include the following:

- a goal decomposition,
- matches of the plans in the goal decomposition against the code,
- bugs,
- differences between the intended function of individual statements and the actual function of these statements,
- possible misconceptions.

This information fits into the following general categories:

- what the programmer did in solving the problem,
- what the programmer was trying to do,
- flaws in his/her knowledge which resulted in problem-solving errors.

These same types of information are found in the student models that many intelligent computer-aided instruction (ICAI) systems generate. All ICAI systems to varying degrees attempt to characterize the student's program-solving process, and the knowledge that underlies this process. ICAI systems tend to differ, however, in how they describe student's knowledge and problem solving. The following will point out some of the differences between PROUST's program interpretations and student models in other systems, and will show how these differences result from characteristics of PROUST's domain.

One major difference between PROUST's diagnostic task and that of other ICAI systems is that PROUST must derive its model of the student from a single program. Systems in the subtraction domain, such as DEBUGGY [5], and in the algebra domain, such as PIXIE [28], tend to work off of a series of student solutions. When multiple problems are analyzed in succession, bug hypotheses derived from one problem can be tested by assigning the student another related problem. The system can then check whether the student solves the new problem as the bug hypothesis would predict. In PROUST's domain, however, it is not practical to assign multiple problems to students and then analyze the bugs afterwards. Each programming problem requires substantial effort on the part of the programmer, and programmers want immediate feedback concerning their program before starting a new one. Thus it is incumbent upon PROUST to analyze the student's errors and give feedback as soon as possible, even if the underlying causes of the bugs are not yet clear to PROUST.

One consequence of the fact that PROUST models the intentions underlying a single program is that it often cannot distinguish what the programmer does in the specific case from what the programmer does in general. If a program has a bug, it may result from an accidental error such as a typographical error, or it may result from a deep-seated misconception. Distinguishing the two explanations requires looking for repeated occurrences of the same bug. If a bug occurs just once, it is likely to be an accidental error; if it occurs consistently, a misconception may be implicated.

PROUST does have one means for checking whether bugs occur systematically; it can check whether the same bug occurs more than once in the same program. The programming problems that we assign students tend to have multiple goals; PROUST can compare the implementation of similar goals to look for systematic errors. If, for example, all initializations are missing from a program, then there is strong evidence that the programmer has a misconception about initializing variables. In simpler domains such as subtraction or algebra there is less opportunity for the same bug to occur more than once within the same problem solution. Thus systematicity of errors within a single program can compensate in part for the lack of a suite of problems to analyze.

When ICAI systems try to model student behavior, they either trace student errors to incorrect problem-solving procedures or to factual misconceptions about the domain. Subtraction systems such as DEBUGGY are good examples of systems that identify incorrect problem-solving procedures. Genesereth's MACSYMA advisor [11] and Clancey's proposed GUIDON-2 system are examples of systems which focus on factual misconceptions. The systems that focus on factual misconceptions, unlike PROUST or DEBUGGY, make strong simplifying assumptions about the student's problem-solving ability. The MACSYMA advisor is a case in point. In the MACSYMA advisor, there is an explicit model of a student problem solver, called MUSER; this same problem solver is used both to solve problems and to model the student solving problems. The advisor assumes that if a student uses MACSYMA incorrectly, it can only be because the student's factual knowledge about how individual MACSYMA commands work is incorrect. The appropriateness of this assumption cannot be assessed without empirical analyses of how novice MACSYMA users actually behave; the advisor appears to be based upon observations of only a handful of students.

In contrast to the MACSYMA advisor and GUIDON-2, PROUST makes fewer assumptions about the student's problem-solving procedure. PROUST assumes that the student's program can be analyzed in terms of goals, but there is no requirement that the goals be decomposed as an expert would. A student might not realize that programmers have to worry about boundary conditions, for example, and still produce a program with a recognizable goal decomposition. Such a student cannot be said to have the same problem-solving procedure as expert programmers, since part of what an expert does is to check systematically for boundary conditions.

This weakening of the assumptions about the student's problem-solving behavior has serious repercussions for error diagnosis. It means that it is not always possible to trace bugs back to the student's factual knowledge. Any given bug may be caused by a factual misconception, a flaw in the student's problem-solving procedure, a failure to follow the problem requirements strictly, or an accidental error. For this reason, bugs must often be described superficially, without reference to underlying cause.

5. Finding the Best Interpretation

The previous sections have described how PROUST generates interpretations for programs. However, PROUST must not only construct interpretations, it must select from among rival interpretations. As we saw in Section 3.3, when PROUST constructs goal decompositions for a program, it constructs a tree of alternative goal decompositions. After the plans are matched, PROUST decides which goal decompositions to explore further. Actually, that is not quite correct; the choice of goal decomposition depends not only upon which plans match, but also how the mismatches are explained in terms of bugs. PROUST is thus comparing program interpretations, not just plan matches. This section describes how PROUST attempts to arrive at the best available interpretation for each program.

Three things are required in order to make sure that PROUST finds the best possible interpretation. First, goals must be selected for decomposition in an order such that interpretation choices are not simply the result of ambiguous plan matches; the methods for selecting goals were described earlier in Section 3.4. Second, evaluation criteria are needed to make sure that each interpretation makes sense. Third, heuristics are required for comparing different interpretations. The discussion in this section will focus on the latter two issues.

5.1. An example of selection among interpretations

In order to show how alternative interpretations for programs can arise, let us return to the example of the Rainfall Problem that was introduced in Fig. 1. The loop from this example is repeated in Fig. 16. One goal decomposition for this loop was discussed in Section 3.1. Now let us consider an alternative goal decomposition for *Sentinel-Controlled Input Sequence*, and see how the program interpretations based upon these two goal decompositions can be compared.

Recall that PROUST knows about four different plans for implementing the goal *Sentinel-Controlled Input Sequence*. Up to now the discussion has focused on one of these plans, the SENTINEL READ-PROCESS REPEAT PLAN. Now let us consider what happens when one of the other plans, the SENTINEL PROCESS-READ WHILE PLAN, is matched against the program. This plan is shown in Fig. 17. In this plan, the first value is input before the main loop is entered; this

```
 8 repeat
 9   writeln ('Enter rainfall');
10   readln;
11   read (Rain);
12   while Rain < 0 do
13     begin
14       Writeln (Rain:0:2, 'is not possible, try again');
15       readln;
16       read (Rain)
17     end;
18
19   while Rain <> 99999 do
20     begin
21       if Rain = 0 then
22         Drydays := Drydays + 1;
23       Totalrain := Totalrain + Rain;
24       if Rain > 0 then
25         Raindays := Raindays + 1;
26       if Highrain < Rain then
27         Highrain := Rain
28     end;
29 until Rain = 99999;
```

Fig. 16. The loop in the example in Fig. 1.

input serves to initialize the variable ?New. The loop itself uses a while statement instead of a repeat statement. The Next step of the loop, which reads in the successive values of ?New, is at the bottom of the loop, instead of at the top, unlike the SENTINEL READ-PROCESS REPEAT PLAN. The part of the loop where the data is processed, labeled Process, is above the Next step. Because the processing step is above the next input step in the loop, no additional *Sentinel Guard* subgoal is required to break out of the loop. In what follows, the SENTINEL PROCESS-READ WHILE PLAN will be abbreviated as S P-R W PLAN, and the SENTINEL READ-PROCESS REPEAT PLAN will be abbreviated as S R-P R PLAN.

The S P-R W PLAN can match the program at least partially, since the program

SENTINEL PROCESS-READ WHILE PLAN

Constants: ?Stop
Variables: ?New
Template:

```
  Initinput:   subgoal Input(?New)
  Mainloop:    while (?New <> ?Stop) do
                 begin
  Process:         ?*
  Next:            subgoal Input(?New)
                 end
```

Fig. 17. Another plan for inplementing *Sentinel-Controlled Input Sequence*.

has a while statement at line 19 which matches the while statement pattern in the plan. Using this plan as a starting point, an interpretation of the loop can be constructed. This interpretation is as follows.

> The while statement in the S P-R W PLAN matches line 19 in the program. The begin-end pair matches lines 20 and 28. The initial *Input* subgoal is combined with the *Input Validation* goal in the problem description, yielding the composite goal *Validated Input*. The VALIDATED PROCESS-READ WHILE INPUT PLAN is a possible implementation of this goal, and it matches lines 10 through 17. The *Input* subgoal in the Next step of the S P-R W PLAN cannot be matched against the program. The student must have omitted implementation of this subgoal. The probable cause is a misconception about iterative inputs; the student may have thought that successive input values would be read in automatically.

The principal differences between the interpretation based upon the S R-P R PLAN and the one based upon the S P-R W PLAN are as follows. The S R-P R PLAN interpretation makes the repeat loop at line 8 the main loop; the S P-R W PLAN makes the while loop at line 19 the main loop. The S R-P R PLAN interpretation regards the input and validation code at lines 10 through 17 as a Next step in the loop; the other interpretation views it as an initialization step. The S P-R W PLAN interpretation gives no interpretation whatsoever to the repeat loop. The S P-R W PLAN interpretation has a missing-input bug, whereas the other interpretation has a while-for-if bug.

Of the various differences mentioned above, the key ones are that the S R-P R PLAN matches more of the program, and does not have any plan components missing. PROUST has an interpretation evaluation heuristic that is relevant to this case:

> Avoid interpretations that leave significant parts of the program uninterpreted.

The rationale for this heuristic is as follows. Each statement is presumed to have a purpose, so if no purpose can be found for a statement, PROUST's goal decomposition may be incorrect. When one interpretation assigns a goal to every statement, and the other interpretation leaves statements without goals, the interpretation which leaves statements without goals is almost certainly incorrect.

The problem with the above evaluation heuristic is that it cannot be applied until the entire goal decomposition is constructed and matched against the program. If PROUST were unable to evaluate interpretations until after they are complete, it would construct large numbers of interpretations, only to find that nearly all of them are wrong. PROUST therefore applies the above heuristic only as a last resort. Instead, it applies other heuristics to detect bad interpretations before the analysis goes too far. The actual heuristic which is applied in

this case is the following:

> Favor interpretations that match more of the program, and have fewer missing plan components.

If this heuristic is applied consistently, PROUST will usually end up with an interpretation which assigns a purpose to as many statements in the program as possible. The heuristic favors the SENTINEL PROCESS-READ REPEAT interpretation because it both matches more of the code and has no missing plan components.

This example illustrates the two kinds of interpretation evaluation processes in PROUST. One kind of evaluation compares one interpretation against other competing interpretations; this is called *differential evaluation* of interpretations. The other kind of evaluation examines the interpretation in isolation, usually after the interpretation is complete; this is called *interpretation assessment*. Both kinds of interpretation will be discussed below.

5.2. Differential evaluation of interpretations

Differentiating program interpretations is closely related to the notion of differential diagnosis in medicine. In performing differential diagnosis a physician compiles a set of etiologies, or causes, which might be relevant to the patient's symptoms. This set of etiologies is called a differential. The physician then tries to narrow down the differential by comparing etiologies against each other, and looking for evidence which confirms one subset of etiologies and disconfirms the others. Eventually the differential is narrowed down to a single diagnosis, the diagnosis which wins out at the end will be demonstrably superior to the alternatives. If, on the other hand, the diagnostician cannot distinguish between competing etiologies, the diagnosis must be considered inconclusive.

The intention-based approach to program analysis lends itself naturally to differential diagnosis. When a goal is selected for analysis, different implementations of the goal are suggested. The set of candidate implementations forms a differential. PROUST then decides which among these implementations fits the program best. Differentiation leads to more robust bug analysis because it allows PROUST to find a wide range of uncommon bugs, bugs which out of context could not be assumed to be present. For example, the while-for-if bugs are hard to identify out of context. Most novice programmers understand the difference between while and if, so PROUST could not presume a while-if confusion without independent evidence. If the while-for-if bug is part of an interpretation that is better than any other interpretation, then the while-for-if diagnosis can be given with more confidence.

Although differential evaluation is desirable, generation of numerous alternative interpretations is undesirable, as it will slow the system down. There are two ways to get around this problem.

- Differentially evaluate partial interpretations.
- Generate just one interpretation; then when the interpretation is complete, perform the differential evaluation, and change the interpretation to reflect the result of the differential evaluation.

The former approach was used in the while-for-if example to determine which plan is used to implement the *Sentinel-Controlled Input Sequence* goal. The latter approach is commonly used in determining whether misconceptions are present. For example, suppose that PROUST finds that an initialization is missing. It can either describe the missing initialization as an accidental bug, or as a manifestation of a misconception about initializations. The two interpretations can be differentiated by checking whether or not initializations were systematically omitted; but such a check cannot be performed until after the interpretation is completed. What PROUST does in this case is to describe the missing initialization independent of cause, and wait until after the interpretation is complete to decide what the cause is likely to be. If it turns out that the initialization bug is systematic, then the bug descriptions of all the missing initializations are changed to indicate a possible misconception.

Differential evaluation of complete interpretations is straightforward, since a substantial amount of information is available for use in comparing interpretations. For example, once the entire program is analyzed, PROUST can check whether or not goals were assigned to every part of the program. Differential evaluation of partial interpretations, on the other hand, is much trickier. The differential interpretation in the previous section is a case in point. When PROUST tries to designate the while statement as the main loop, it is leaving the repeat statement uninterpreted. At this point it is not known whether or not there is some other goal pending on the goal agenda that might account for repeat statement. PROUST is thus making the differentiation on the basis of incomplete information, and must rely upon heuristic comparisons of the partial interpretations. The following discussion will focus on differential evaluation of partial interpretations, and show why PROUST's heuristic approach is usually successful.

5.2.1. *Differential evaluation of partial interpretations*

Differential evaluation of partial interpretations is performed at each branch point in the interpretation tree where plans are selected. As described earlier, whenever a goal is selected for decomposition a number of branches of the tree are constructed, each branch ending in a plan. The plans are then matched in parallel. If some of the plans have mismatches, plan-difference rules are applied to explain the mismatches. If a mismatch cannot be explained, the offending plan component is presumed to be missing from the program. Once plan matching and plan-difference analysis is complete, the differential evaluation process can begin.

The first step in choosing among plans is to filter out those which have implausibly many components missing. We consider it plausible that a component be missing if in our empirical studies of novice programs we observe that novice programmers occasionally leave the component out. For example, it is plausible for the initialization component of any plan to be missing. There exists a set of rules in PROUST's knowledge base, similar to plan-difference rules, which trigger when plan components are found to be missing, and indicate what bugs might cause them. If no rules can account for why a plan has missing components, the plan is thrown out.[4] All that remain are plans that can be mapped onto the code in some plausible way. If all plans for a given goal are thrown out, then PROUST concludes that the program does not implement the goal.

The second step in choosing a plan is to count the number of components that matched in each plan, and select those with the greatest number of matched components. In other words, PROUST selects the greediest match. It is this criterion which determines that the repeat loop is the main loop in the program in Fig. 16. Greedy selection works for two reasons. First, not many large plans pass the first selection step; those which remain are probably the right matches. Second, if greedy selection picks the wrong plan, the selection error will probably be discovered later, when a plan implementing a different goal is found to match the same code. When two plans implementing different goals match the same code, one or the other of the two plans is likely to be matched incorrectly. Our ultimate aim is to give PROUST the capability of then deciding between the two matches to the same code, and choosing different plans to remove the conflict.

If two plans have the same number of matching components, they are further differentiated by counting the number of misconceptions that are suggested by the plan-difference rules. Interpretations which do not suggest misconceptions are favored over those which do.

If these criteria fail to identify a unique interpretation as the best match, PROUST puts the analysis of the goal aside, and selects a different goal for decomposition. PROUST will then reconsider the analysis of the goal later, after other goals have been analyzed. This allows PROUST to rely upon Occam's razor in selecting interpretations. PROUST assumes that each part of the program serves a unique purpose, unless the goal decomposition explicitly dictates that two goals are being combined. If part of the program clearly implements a particular goal, then it can be assumed that it does not match any other goals. Interpretations of the other goals are restricted to those parts of the program which have not previously been interpreted. The following

[4] These missing-component rules were first introduced before PROUST's differential evaluation mechanism was in place. It now appears that these rules can be replaced by more general heuristics that can be incorporated into the differential evaluation mechanism.

example serves to illustrate. Suppose that a student confuses running total updates and counter updates, writing Sum := Sum + 1 instead of Sum := Sum + New. In such cases it may be hard to tell which update is supposed to be the running total update, since there may be several increment statements in the program. If PROUST postpones the decision and matches the *Count* goals instead, then through a process of elimination PROUST can eventually identify the buggy running total update.

5.3. Interpretation assessment

In spite of PROUST's attempts to select the best interpretation for programs, it sometimes makes mistakes. Programs sometimes have bugs that PROUST's plan-difference rules cannot recognize. The programmer may use variables in inconsistent ways, giving PROUST false expectations about the bindings of pattern variables in plans. In order to make sure that no mistake was made in interpreting the program, PROUST examines the program interpretation as a whole once analysis is complete, looking for evidence that the interpretation is incorrect. If such evidence is found, questionable parts of the interpretation are deleted, to guard against giving an incorrect bug report to the student.

The most obvious indication of interpretation failure is that very few of the goals in the problem description were successfully mapped onto the student's code. PROUST requires that interpretations be found for a significant fraction of the goals in the problem description. If this fails to happen, the analysis is aborted, and no bugs are reported to the student.

Even if most of the goals are mapped onto the code, the analysis is possibly flawed if some of the code could not be analyzed, and some of the goals could not be mapped onto the code. When this happens, PROUST performs bottom-up analysis on the interpreted code, to see what kind of function it performs, and then compares this against the outstanding goals. If the function of some of the code is close to one or more of the outstanding goals, then the interpretation is probably in error. The interpretation is classified by PROUST as a partial analysis. Bugs which may be in error because of the misinterpretation, such as complaints about unimplemented goals, are deleted from the bug report that is presented to the student. This process is described in more detail in [16].

6. Empirical Evaluation of PROUST

The bottom-line issue in evaluating the work that has gone into PROUST is whether or not it has resulted in an effective tool for finding novice bugs. This cannot be determined by observing PROUST's behavior on a few student programs; instead, PROUST must be tested on hundreds of student programs, in a variety of situations. The results of some of these empirical tests are presented below. Further results are published elsewhere [16].

6.1. Results on the Rainfall Problem

Table 1 shows the results of running PROUST off-line on a corpus of 206 different solutions of the Rainfall Problem. The percentage of programs which are analyzed completely is 81%. PROUST's analysis is complete if a complete interpretation was generated, in which interpretation assessment could not find any inconsistencies. 15% of the programs were analyzed partially, meaning that a substantial part of the program was analyzed, but the interpretation was incomplete or inconsistent. In 4% of the cases the analysis was aborted, either because hardly any goals were successfully analyzed or because some construct such as goto, which PROUST is not prepared to analyze, appears in the program. When PROUST analyzes programs completely, it identifies 94% of the bugs, as determined by hand analyses of the same programs. Note that these were not 94% of the bugs that we expected PROUST to detect; they were 94% of *all semantic and logical errors*. At the same time there are a certain number of false alarms, i.e., cases where PROUST either misinterpreted a bug or flagged a bug which did not really exist. Most of these false alarms result from misinterpretations of the programs' goal decompositions, often because an unusual plan or bug was present. Further extension and generalization of PROUST's knowledge base would be required in order to reduce the occurrence of false alarms. The Rainfall Problem was subsequently assigned to another PASCAL class, and was tested on line. The subsequent results were comparable, with 70% of the programs receiving full analysis, and 98% of the bugs in these programs correctly recognized.

Table 1
Results of running PROUST on the Rainfall Problem

Total number of programs:	206	
Number of programs with bugs:	183	(89%)
Number of programs without bugs:	23	
Total number of bugs:	795	
Number of programs receiving full analyses:	167	(81%)
Total number of bugs:	598	(75%)
Bugs recognized correctly:	562	(94%)
Bugs not recognized:	36	(6%)
False alarms:	66	
Number of programs receiving partial analyses:	31	(15%)
Total number of bugs:	167	(21%)
Bugs recognized correctly:	61	(37%)
Bugs not reported:	106	(63%)
False alarms:	20	
Number of programs PROUST did not analyze:	9	(4%)
Total number of bugs:	32	(4%)

6.1.1. *Results on a different problem*

In a further test, PROUST was tested on a different programming problem, called the Bank Problem.

Bank Problem. Write a PASCAL program that processes three types of bank transactions: withdrawals, deposits, and a special transaction that says: no more transactions are to follow. Your program should start by asking the user to input his/her account id and his/her initial balance. Then your program should prompt the user to input

(1) the transaction type,

(2) if it is an END-PROCESSING transaction, the program should print out (a) the final balance of the user's account, (b) the total number of transactions, (c) the total number of each type of transaction, and (d) the total amount of the service charges, and stop;

(3) if it is a DEPOSIT or a WITHDRAWAL, the program should ask for the amount of the transaction and then post it appropriately.

Use a variable of type CHAR to encode the transaction types. To encourage saving, charge the user 20 cents per withdrawal, but nothing for a deposit.

In this problem, the students are required to write a program which behaves similarly to an automatic bank teller machine. The program is supposed to input a series of deposit and withdrawal commands, followed by an end-processing command. The user's account balance is updated according to the amount of each deposit and withdrawal. At the end of the program a summary of the transactions is printed.

Table 2 shows PROUST's current performance on the Bank Problem. The frequency of completed analyses is much lower than in the case of the Rainfall Problem; it analyzed 50% of the programs, as opposed to 81% on the Rainfall Problem. PROUST's performance on the completely analyzed programs is almost as good as it is on completely analyzed solutions of the Rainfall Problem. 91% of the bugs in the Bank Problem solutions were correctly identified, compared with 94% of the bugs in the Rainfall Problem solutions. The incidence of false alarms, however, is relatively high; there were 41 false alarms in the completely analyzed Bank Problem solutions, compared with 211 total bugs in the same group of programs. Four programs were omitted from analysis because they were very far removed from an expected solution, to such an extent that they were not considered a fair test of PROUST.

There appear to be several reasons why PROUST's performance on the Bank Problem is less than that on the Rainfall Problem. First, the problem requires that more goals be satisfied than the Rainfall Problem requires; this was an intended feature of the problem. Other problems with the Bank Problem, however, were unanticipated. For one thing, many more of the goals of the Bank Problem were left implicit. For example, the problem statement says

Table 2
Results of running PROUST on the Bank Problem

Total number of programs analyzed:	64	
Total numbers of bugs	420	
Number of programs receiving full analyses:	32	(50%)
Total number of bugs:	211	(50%)
Bugs recognized correctly:	191	(91%)
Bugs not reported:	20	(9%)
False alarms:	41	
Number of programs receiving partial analyses:	26	(41%)
Total number of bugs:	168	(40%)
Bugs recognized correctly:	56	(33%)
Bugs not reported:	112	(67%)
False alarms:	24	
Number of programs PROUST did not analyze:	6	(9%)
Total number of bugs:	41	(10%)
Number of programs omitted from analysis:	4	

nothing about what to do if the balance becomes less than zero. Some solutions had no checks for negative balance, some checked the balance only after the last transaction is complete, and some checked the balance after each transaction. PROUST did not generate all of these different goal decompositions, so it failed to interpret some programs. Another difference between the two problems is that the Bank Problem provides no explicit cues to disambiguate plan matching. The Rainfall Problem states explicitly that the sentinel value is 99999; the plans for matching *Sentinel-Controlled Input Sequence* therefore usually match unambiguously, since there is only one loop in a given solution which tests for 99999. The Bank Problem, on the other hand, does not state specifically which commands are to be used to indicate deposit, withdrawal, or end-processing transactions. There is therefore a much greater risk of ambiguous matches, and consequently of misinterpretations of the program. It appears likely that a more detailed problem statement, in which the transaction commands were listed explicitly, would have improved PROUST's performance.

7. Concluding Remarks and Future Directions

This article has claimed that accurate debugging of novice programs requires an understanding of the intentions underlying programs. Without an understanding of the programmer's intentions, many bugs cannot be detected, and those that can be detected cannot be localized and explained. In order to diagnose bugs effectively, one needs knowledge both of what the program is intended to do and how it is intended to do it. Since the intentions underlying each program may be different, the precise intentions of the programmer must

be inferred from the buggy program as it is being analyzed. The key to doing this is to start with a description of the problem being solved, and to use programming knowledge to predict possible ways in which the problem might be solved. In most cases the programmer's intentions can be related to the predictions. A program called PROUST was built which uses this approach to diagnose bugs in novice programs.

Although significant results have been achieved so far with PROUST, much more work remains to be done. One of the most immediate needs at this time is to test PROUST on a wider range of programming problems. Now that an acceptable level of performance has been achieved on two programming problems, the time has come to try to generalize and extend PROUST's knowledge. In addition, detailed feedback from novice users would be helpful, to ensure that PROUST's goal decompositions accurately reflect the novices' intentions.

Until PROUST is coupled with a tutoring module, PROUST's ability to diagnose programming errors will remain limited to the information that is extractable from the buggy programs themselves. A tutoring component would be able to ask students questions in order to select between alternative explanations of bugs. It would allow PROUST to refine its model of the student's abilities, to make more explicit its model of the student's knowledge and problem-solving skills. This in turn would allow PROUST to make more accurate predictions about the students' intentions, and derive a deeper understanding of the students' bugs. Such a tutor is currently in the process of being developed [20, 22].

A version of PROUST should be developed for a different programming language, such as ADA or LISP. Building such a system would help determine the generality of PROUST's approach, and would provide further insights into the kinds of knowledge that programmers use in solving problems. It would also be useful to apply PROUST's approach to other domains, in order to demonstrate the generality of intention-based analysis as a means for identifying and correcting bugs. There are already some promising results in this direction: Sebrechts has taken a stripped-down version of PROUST, called MICRO-PROUST [18], and adapted it to the domain of statistics. The resulting system, GIDE, has undergone preliminary tests with statistics students [24]. PROUST's approach should be useful in a variety of domains where students are given sets of goals to solve, and must combine plans in order to construct a solution which achieves these goals.

ACKNOWLEDGMENT

I would like to thank Elliot Soloway, my advisor, for insightful suggestions regarding this research, and for comments on an earlier draft of this paper. I would also like to thank Bill Clancey for his comments regarding this paper, and for his help in clarifying some of the ideas presented here. Bill Swartout and Bob Neches also made helpful suggestions regarding the paper.

This work was co-sponsored by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number, No. 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

REFERENCES

1. Adam, A. and Laurent, J.-P., LAURA: A system to debug student programs, *Artificial Intelligence* **15** (1980) 75–122.
2. Barnard, D.T., A survey of syntax error handling techniques, Tech. Rept., University of Toronto, Toronto, Ont. (1976).
3. Bonar, J., Ehrlich, K. and Soloway, E., Collecting and analyzing on-line protocols from novice programmers, *Behav. Res. Methods Instrumentation* **14** (1982) 203–209.
4. Brown, J.S. and Burton, R.R., Diagnostic models for procedural bugs in mathematics, *Cognitive Sci.* **2** (1978) 155–192.
5. Burton, R.R., Diagnosing bugs in a simple procedural skill, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
6. Conway, R.W. and Wilcox, T.R., Design and implementation of a diagnostic compiler for PL/I, *Commun. ACM* **16** (1973) 169–179.
7. Davis, R., Shrobe, H., Hamscher, W., Weickert, K., Shirley, M. and Polit, S., Diagnosis based on description of structure and function, in: *Proceedings AAAI-82*, Pittsburgh, PA (1982) 137–142.
8. Eisenstadt, M., Prospective zooming: A knowledge based tracing and debugging methodology for logic programming, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 717–719.
9. Farrell, R.G., Anderson, J.R. and Reiser, B.J., An interactive computer-based tutor for LISP, in: *Proceedings AAAI-84*, Austin, TX (1984) 106–109.
10. Fosdick, L.D. and Osterweil, L.J., Data flow analysis in software reliability, *Comput. Surv.* **8** (1976) 305–330.
11. Genesereth, M.R., Automated consultation for complex computer systems, Ph.D. Thesis, Harvard University, Cambridge, MA (1978).
12. Genesereth, M.R., The role of plans in intelligent teaching systems, in: D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
13. Graham, S.L. and Rhodes, S.P., Practical syntactic error recovery in compilers, *Commun. ACM* **18** (11) (1975).
14. Harandi, M.T., Knowledge-based program debugging: A heuristic model, in: *Proceedings 1983 SOFTFAIR* (1983).
15. James, E.B. and Partridge, D.P., Adaptive correction of program statements, *Commun. ACM* **16** (1) (1973).
16. Johnson, W.L., *Intention-Based Diagnosis of Novice Programming Errors* (Morgan Kaufmann, Los Altos, CA, 1986).
17. Johnson, W.L., Draper, S. and Soloway, E., Classifying bugs is a tricky business, in: *Proceedings NASA Workshop on Software Engineering* (to appear).
18. Johnson, W.L. and Soloway, E., PROUST: An automatic debugger for Pascal programs, in: *Artificial Intelligence and Instruction: Applications and Methods* (Addison-Wesley, Reading, MA, 1986).
19. Johnson, W.L., Soloway, E., Cutler, B. and Draper, S., Bug collection I, Tech. Rept. 296, Department of Computer Science, Yale University, New Haven, CT (1983).
20. Littman, D.C., Pinto, J. and Soloway, E., An analysis of tutorial reasoning about programming bugs, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 320–326.
21. Murray, W.R., Automatic program debugging for intelligent tutoring systems, Ph.D. Thesis, University of Texas, Austin, TX (1986).

22. Rich, C., A formal representation for plans in the programmer's apprentice, in: *Proceedings IJCAI-81*, Vancouver, BC (1981) 1044–1052.
23. Sack, W. and Soloway, E., From MENO to PROUST to CHIRON: AI design as iterative engineering; Intermediate results are important!, in: *Proceedings Invited Workshop on Computer-Based Learning Environments*, Pittsburgh, PA (1988).
24. Sebrechts, M., Schooler, L., LaClaire, L. and Soloway, E., Computer-based interpretations of students' statistical errors: A preliminary empirical analysis of GIDE, in: *Proceedings 8th National Educational Computing Conference*, Philadelphia, PA (1987).
25. Sedlmeyer, R.L. and Johnson, P.E., Diagnostic reasoning in software fault localization, in: *Proceedings SIGSOFT Workshop on High-Level Debugging*, Asilomar, CA (1983).
26. Shapiro, D.G., Sniffer: A system that understands bugs, Tech. Rept. AI Memo 638, MIT Artificial Intelligence Laboratory, Cambridge, MA (1981).
27. Shapiro, E., *Algorithmic Program Debugging* (MIT Press, Cambridge, MA, 1982).
28. Sleeman, D., A rule directed modelling system, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligent Approach* (Tioga, Palo Alto, CA, 1983).
29. Soloway, E. and Ehrlich, K., Empirical investigations of programming knowledge, *IEEE Trans. Software Eng.* **10** (5) (1984).
30. Soloway, E., Rubin, E., Woolf, B., Bonar, J. and Johnson, W.L., MENO-II: An AI-based programming-tutor, *J. Comput.-Based Instruction* **10** (1) (1983).
31. Swartout, W., The Gist behavior explainer, in: *Proceedings AAAI-83*, Washington, DC (1983). (Also available as ISI/RR-83-3.)
32. Teitelman, W., *INTERLISP Reference Manual* (1978).
33. Wertz, H., Stereotyped program debugging: An aid for novice programmers, *Int. J. Man-Mach. Stud.* **16** (1982) 379–392.