Intention-Based Diagnosis of Programming Errors

W. Lewis Johnson - Elliot Soloway

Yale University Computer Science Department
New Haven, Ct. 06520

Abstract

PROUST is a system which identifies the non-syntactic bugs in novices' programs and provides novices with help as to the misconceptions under which they were laboring that caused the bugs. In this paper we will discuss the methods which PROUST uses to identify and diagnose non-syntactic bugs. Key in this enterprise is PROUST's ability to cope with the significant variability exhibited by novices' programs: novice programs are designed and implemented in a variety of different ways, and usually have numerous bugs. We argue that diagnostic techniques that attempt to reason from faulty behavior to bugs are not effective in the face of such variability. Rather, PROUST's approach is to construct a causal model of the programmer's intentions and their realization (or non-realization) in the code. This model serves as a framework for bug recognition, and allows PROUST to reason about the consequences of the programmer's decisions in order to determine where errors were committed and why.

1. Introduction

We have been constructing a system, PROUST, that can identify the non-syntactic bugs in novice's programs and provide students with help in resolving the misconceptions that caused the bugs. In this paper we will discuss the principal techniques PROUST uses to identify and explain non-syntactic bugs.

PROUST's analysis techniques were designed to cope with the key feature of our domain: the high degree of variability in novice programs. Novice programmers often have misconceptions about programming language syntax and semantics, resulting in large numbers of seemingly bizarre bugs. They also lack the expert's knowledge about how to analyze program specifications and design and implement algorithms. The result is that the intentions underlying novice programs, and the methods used for realizing these intentions, tend to vary greatly. We present an approach to error diagnosis which integrates identifying program errors with discovering the programmer's intentions. In this view, bug diagnosis involves reasoning in the space of intentions as well as in the space of program behavior. This contrasts with conventional fault diagnosis methods which do not take intentions into account, or which fail to distinguish between intended program behavior and actual program behavior. The intentional model that PROUST constructs provides a framework for testing bug hypotheses, and for comparing alternative hypotheses using differential analysis. Causal reasoning about the programmer's intentions makes it possible to determine which of the programmer's intentions is faulty and why.

In what follows, we first present arguments in support of the intention-based approach, followed by a more detailed description of PROUST's approach to error diagnosis. Results of empirical tests of PROUST with novice programmers will then be presented.

2. Three Approaches to Diagnosis

We can identify at least three types of diagnostic reasoning techniques which might be applicable to bug diagnosis: classificatory reasoning about symptoms, causal reasoning about behavior, and intention modeling.

- In classificatory reasoning about symptoms, the diagnostician knows what classes of symptoms different classes of faults exhibit. The diagnostician extracts important facts from the findings, uses them to suggest types of faults which might explain the findings, and then does further analysis in order to refine the diagnosis. The diagnostician's knowledge about the domain can thus be boiled down to a collection of classificatory rules relating symptoms to disease classes. Medical diagnosis systems tend to depend particularly heavily upon classificatory reasoning [15, 3]. Classificatory approaches to program debugging have also been attempted [6].
- Causal reasoning about behavior uses an understanding of the structure and function of a system and its components to identify faults which are responsible for faulty behavior [11, 4, 5, 14]. In program debugging the causal reasoning usually takes the form of analysis of control and data flow. Causal reasoning is useful for diagnosing errors in domains involving a degree of complexity and variability, where empirical associations between faults and symptoms are unavailable or inconclusive.
- Intention modeling is necessary, however, when the

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, Nr 154-492. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

programmer's intentions are in odds with what the programming problem requires. Here, the programmer's view of the problem must be determined and then tied to the manner in which it has/has not been realized. Knowledge of intentions assists the diagnostic process in several ways. Predictions derived from the intention model enable top-down understanding of buggy code, so that diagnosis is not thrown off when bugs obscure the code's intended behavior. The right fix for each bug can be found [8]. Finally, causal reasoning about the implications of the implementor's intentions makes it possible to test bug hypotheses by looking at other parts of the program and verifying the implications of bug hypotheses.

We will argue that neither classificatory reasoning about symptoms nor causal reasoning about behavior is adequate for fault diagnosis in novice programs. Rather, accurate bug diagnosis depends upon intention modeling.

3. Intention-Based Diagnosis vs. Other Approaches: An Example

We will first walk through an example of bug diagnosis in order to contrast the intention-based approach to the other diagnostic approaches. Further details of how intention-based diagnosis works will be provided in subsequent sections.

Figure 1 shows a programming problem, to compute the average of a series of inputs, and an actual buggy student solution. Instead of reading a series of inputs and averaging them, this program reads a single number `New`, and outputs the average of all the values between `New` and 99999, *i.e.*, `(New+99999)/2`. We believe that the error in this program is that the student wrote `New := New+1` at line 12 instead of `Read(New)`, as indicated by PROUST's output, which is shown in the figure. This bug is probably the result of a programming misconception: novices sometimes overgeneralize the counter increment statement and use it as a general mechanism for getting the next value.

A symptom classification approach would make use of general heuristic rules for relating symptoms to causes. Example rules might be the following:

- If a program terminates before it has read enough input, it may have an input loop with a faulty exit test.
- If a program outputs a value which is too large, check the line that computes the value and make sure that it is correct.

Neither of these rules addresses the true cause of the bug; instead of focusing on the way new values are generated, one rule focuses on the exit test, and the other focuses on the average computation. In general, many different program faults can result in the same symptoms, so knowledge of the symptoms alone is insufficient for distinguishing faults.

The principal diagnostic methods which employ causal analysis of behavior are symbolic execution, canonicalization, and troubleshooting. If we followed a symbolic execution paradigm, as in PUDSY [10], we would go through the following sequence of steps: 1) use causal knowledge of program semantics to derive a formula describing the output of the program, 2) compare it against a description of what the program is supposed to do, in order to identify errors, and then 3) trace the erroneous results back to the code which generated them. In the example in Figure 1, we would determine that the program computes `New/2+49999.5`, compare this against what it should compute, namely `(ΣNew)/count(New)`, then examine the parts of the program which compute the erroneous parts of the formula. The main problem here is that it is hard to compare these two expressions and determine which parts are wrong. This requires knowledge of which components of the first expression correspond to which components of the second expression. Expression components correspond only if their underlying intentions correspond. Thus some knowledge of the programmer's intentions is necessary in order for the symbolic execution approach to generate reliable results.

Problem: Read in numbers, taking their sum, until the number 99999 is seen. Report the average. Do not include the final 99999 in the average.

```
 1   PROGRAM Average( input, output );
 2   VAR Sum, Count, New: INTEGER;
 3       Avg: REAL;
 4   BEGIN
 5     Sum := 0;
 6     Count := 0;
 7     Read( New );
 8     WHILE New<>99999 DO
 9       BEGIN
10         Sum := Sum+New;
11         Count := Count+1;
12         New := New+1
13       END;
14     Avg := Sum/Count;
15     Writeln( 'The average is ', avg );
16   END;
```

PROUST output:

It appears that you were trying to use line 12 to read the next input value. Incrementing NEW will not cause the next value to be read in. You need to use a READ statement here, such as you use in line 7.

Figure 1: Example of analysis of a buggy program

Canonicalization techniques [1] translate the student's program into a canonical dataflow representation and compare it against an idealized correct program model. Again the aim is to determine intentions by comparison. Such comparisons are easier to make, but only if the student's intended algorithm is the same as the model algorithm. Non-trivial programming problems can be solved in any of a number of ways. Thus an approach which compares against a single model solution cannot cope with the variability inherent in programming.

Troubleshooting approaches suffer from similar difficulties as symbolic execution. In program troubleshooting, the user is expected to describe the specific symptoms of the fault, rather than give a description of the intended output. The system then traces the flow of information in the program to determine what might have caused the symptoms. In this example the symptom is that the program computes `New/2+49999.5`. We have already seen in the case of symbolic execution that this information alone is not sufficient to pinpoint the bug.

Because of the problems that analysis of symptoms and behavior pose in debugging, a number of implementors of bug diagnosis systems have augmented these techniques with recognizers for stereotypic programming plans [13, 12, 16]. This is an attempt at determining the intentions underlying the code; by recognizing a plan we can infer the intended function of the code which realizes the plan, which in turn helps in localizing bugs. Unfortunately plan recognition by itself is not adequate for inferring intentions. First, bugs can lead to plan recognition confusions. In the example, the loop looks like a counter-controlled iteration; diagnosing the bug requires the realization that the loop was not intended to be counter controlled. Second, bugs may arise not in plans themselves, but in the way that they interact or in the manner that the programmer has employed them. To a certain extent one can determine the interactions of plans in a program by analyzing the flow of information among the plans. However, we will consider examples in the next section where the intended plan interactions and the actual plan interactions are different. In such cases a better understanding of the programmer's intentions is needed than what plan recognition alone can provide.

In contrast to these other approaches, the intention-based approach attempts to construct a coherent model of what the programmer's intentions were and how they were realized in the program. Instead of simply listing the plans which occur in the program, one must build a *goal structure* for the program; *i.e.*, one must determine what the programmer's goals were, and how he/she went about realizing those goals, using plans or some other means. This is accomplished in PROUST as follows. PROUST is given an informal description of what the program is supposed to do. It then makes use of a knowledge base of relations between goals and plans, on one hand, and rules about how goals combine and interact, on the other hand, to suggest possible goal structures for the program. The goal structure that fits the best suggests that the `Read` at line 7 satisfies the goal of initializing the `WHILE` loop; the loop is organized to process each value and then read the next value at the bottom of the loop, in a process-n-read-next-n fashion. Lines 5 and 10 are responsible for totaling the inputs, lines 6 and 11 are responsible for counting them, and line 14 computes the average. Given this model, there is no role for the `New := New+1` to serve other than as a means to read the next value inside the loop. This leads directly to the conclusion that the student has overgeneralized the use of counter increment statements.

4. Examples of Intention-Based Diagnosis

Program analysis in PROUST involves a combination of shallow reasoning for recognizing plans and bugs, and causal reasoning about intentions. The relative importance of each kind of reasoning depends upon the complexity of the program's goal structure and the extent to which PROUST must analyze the implications of the programmer's design decisions and misconceptions to determine what bugs they cause. We will show how PROUST reasons about programs and bugs by way of a series of examples. The first program example has no bugs, and has a simple goal structure; accordingly, the reasoning processes involved in understanding it are primarily shallow. The next example has bugs, and the goal structure is somewhat more complex; although the bugs are discovered via shallow recognition tactics, a greater amount of reasoning about intentions is required to construct the right goal structure and test the bug hypothesis. In the third example the programmer's intentions are not reflected directly in the code, so PROUST must use knowledge about goal interactions to hypothesize and differentiate possible intention models for the program.

4.1. Shallow reasoning about correct programs

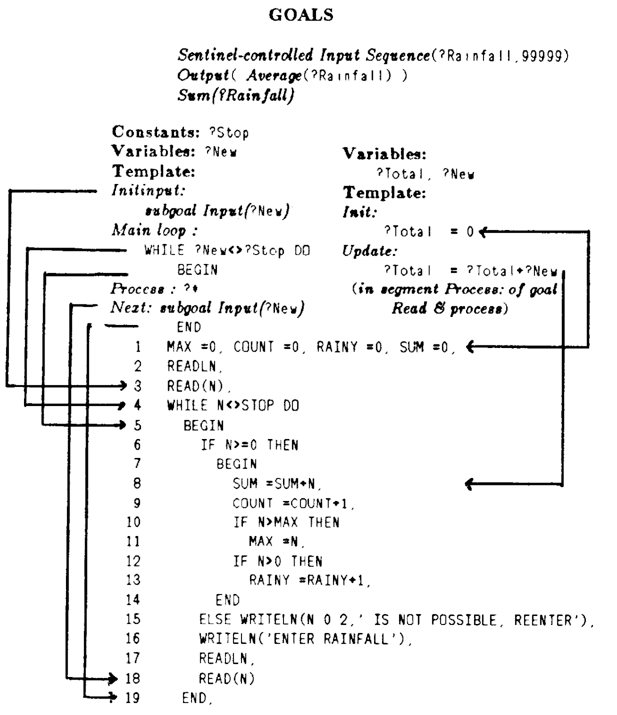
Figure 2 shows a typical introductory programming problem. Figure 3 shows the plan analysis of a straightforwardly correct solution, *i.e.*, one in which the intentions are correct and the program is implemented in accordance with rules of programming discourse [17]. PROUST is supplied with a description of the programming problem, shown in Figure 4, which reflects the problem statement which the students are given. This description is incomplete, in that details are omitted and terms are used which must be defined by reference to PROUST's knowledge base of domain concepts. PROUST derives from the problem description an agenda of goals which must be satisfied by the program. PROUST must go through a process of building hypothetical goal structures using these goals, and relating them to the code. We call this process constructing interpretations of the code.

The goal structure for a given program is built by selecting goals to be processed, determining what plans might be used to implement these goals, and then matching them against the code. Let us consider what happens to the *Sentinel-Controlled Input Sequence* goal. This goal specifies that input should be read and processed until a specific sentinel value is read. PROUST must determine what plans might be used to realize this goal. PROUST has a knowledge base of typical programming plans, and another knowledge base of programming goals. Each plan is indexed according to the goals it can be used to implement, and each goal linked to plans and/or collections of subgoals which implement it. PROUST retrieves from these databases several plans which implement the goal. One of these, the SENTINEL-CONTROLLED PROCESS-READ WHILE PLAN, is shown in Figure 3. This plan specifies that there should be a `WHILE` loop which reads and processes input in a process-n-read-next-n fashion; we saw a buggy instance of this plan in the program in Figure 1.

Noah needs to keep track of rainfall in the New Haven area in order to determine when to launch his ark. Write a program which he can use to do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 2: The Rainfall Problem

Each plan consists of a combination of statements in the target language, Pascal, and subgoals. The syntax and semantics of plans, and the methods used for matching them, are discussed in detail in [7]. Matching the plan against the code involves 1) finding statements which match the Pascal part of the plan, and 2) selecting and matching additional plans to implement the plan's subgoals. For example, the `WHILE` loop at line 4 matches the Pascal part of the plan. This plan also has two subgoals, both *Input* goals. The plans for implementing these goals which match the code are both READ



```
                         GOALS

          Sentinel-controlled Input Sequence(?Rainfall,99999)
          Output( Average(?Rainfall) )
          Sum(?Rainfall)

Constants: ?Stop
Variables: ?New                    Variables:
Template:                              ?Total, ?New
Initinput:                         Template:
    subgoal Input(?New)            Init:
Main loop :                            ?Total  = 0
    WHILE ?New<>?Stop DO           Update:
       BEGIN                           ?Total  = ?Total+?New
Process : ?*                       (in segment Process: of goal
Next: subgoal Input(?New)              Read & process)
       END
   1  MAX =0, COUNT =0, RAINY =0, SUM =0,
   2  READLN,
   3  READ(N),
   4  WHILE N<>STOP DO
   5    BEGIN
   6      IF N>=0 THEN
   7        BEGIN
   8          SUM =SUM+N,
   9          COUNT =COUNT+1,
  10          IF N>MAX THEN
  11            MAX =N,
  12          IF N>0 THEN
  13            RAINY =RAINY+1,
  14        END
  15      ELSE WRITELN(N 0 2,' IS NOT POSSIBLE, REENTER'),
  16      WRITELN('ENTER RAINFALL'),
  17      READLN,
  18      READ(N)
  19    END,
```

Figure 3: Plan Recognition

SINGLE VALUE plans, *i.e.*, READ statements which read single values. The SENTINEL-CONTROLLED PROCESS-READ PLAN thus matches the program exactly, so this plan is incorporated into the goal structure.

```
DefProgram Rainfall;

DefObject ?Rainfall:DailyRain Type ScalarMeasurement;

Sentinel-Controlled Input Sequence( ?Rainfall:DailyRain, 99999 );
Input Validation( ?Rainfall:DailyRain,
                  ?Rainfall:DailyRain<0 )
Output( Average( ?Rainfall:DailyRain ) );
Output( Count( ?Rainfall:DailyRain ) );
Output( Count( ?Rainfall:DailyRain
               s.t. ?Rainfall:DailyRain>0 ) );
Output( Maximum( ?Rainfall:DailyRain ) );
```

Figure 4: Representation of the Rainfall Problem

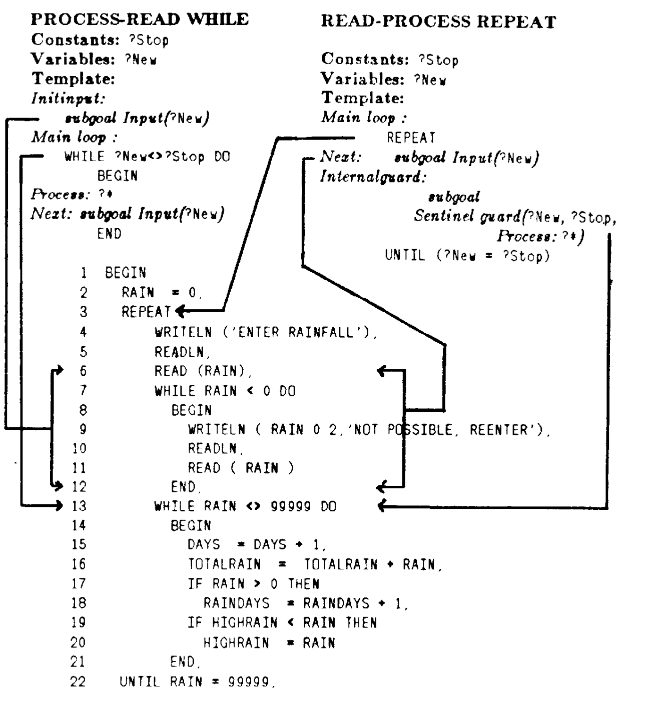
PROUST continues selecting goals from the agenda and mapping them to the code, until every goal has been accounted for. This involves some analysis of implications of plans. For example, the choice of plan for computing the *Average* goal implies that a *Sum* goal be added to the goal agenda. This is in turn implemented using a RUNNING TOTAL PLAN, shown in the figure. However, in a program such as this relatively little work is involved in manipulating the goal agenda; most of the work in understanding this program is in the plan recognition.

PROUST is thus able to analyze straightforwardly correct programs primarily using shallow plan recognition techniques. This is not a surprising result. We have argued elsewhere [17] that programmers make extensive use of stereotypic plans when writing and understanding programs. Furthermore, we have evidence that novice programmers acquire plans early on [2]. We encourage this by including plans in our introductory programming curriculum. We can therefore assume that plans will play a major role in the construction of the programs that PROUST analyzes. We assume furthermore that if a programmer uses plans correctly, and if they fit together into a coherent design, then the functionality of the plans corresponds closely to the programmer's intentions.

4.2. Differentiating program interpretations

We will now look at an example which involves integrating bug recognition into the process of constructing program interpretations. Recall that the problem statement in Figure 2 requires that all non-negative input other than 99999 should be processed. However, the program in Figure 5 goes into an infinite loop as soon as a non-negative value other than 99999 is read. The reason is that the WHILE statement at line 13 should really be an IF statement. The programmer is probably confused about the semantics of nested WHILE statements, a common difficulty for novice Pascal programmers [9]. Otherwise the the loop is constructed properly. Apparently programmer understands how WHILE loops work when the body of the loop is straight-line code, but is confused about how multiple tests are integrated into a single loop. We will show how PROUST develops this interpretation for the program.

The bug in this example is encountered while PROUST is processing the *Sentinel-controlled input sequence* goal. Two plans implementing this goal match the code: the SENTINEL PROCESS-READ WHILE PLAN, which we saw in the previous example, and the SENTINEL READ-PROCESS REPEAT PLAN. The WHILE loop plan matches the loop starting at line 13, while the REPEAT loop plan matches the loop starting at line 3.



```
PROCESS-READ WHILE                 READ-PROCESS REPEAT
Constants: ?Stop
Variables: ?New                    Constants: ?Stop
Template:                          Variables: ?New
Initinput:                         Template:
    subgoal Input(?New)            Main loop :
Main loop :                               REPEAT
    WHILE ?New<>?Stop DO           Next:     subgoal Input(?New)
       BEGIN                       Internalguard:
Process: ?*                                  subgoal
Next: subgoal Input(?New)                 Sentinel guard(?New, ?Stop,
       END                                        Process: ?*)
                                          UNTIL (?New = ?Stop)

   1  BEGIN
   2    RAIN  = 0,
   3    REPEAT
   4        WRITELN ('ENTER RAINFALL'),
   5        READLN,
   6        READ (RAIN),
   7        WHILE RAIN < 0 DO
   8          BEGIN
   9            WRITELN ( RAIN 0 2,'NOT POSSIBLE, REENTER'),
  10            READLN,
  11            READ ( RAIN )
  12          END,
  13        WHILE RAIN <> 99999 DO
  14          BEGIN
  15            DAYS  = DAYS + 1,
  16            TOTALRAIN  =  TOTALRAIN + RAIN,
  17            IF RAIN > 0 THEN
  18              RAINDAYS  = RAINDAYS + 1,
  19            IF HIGHRAIN < RAIN THEN
  20              HIGHRAIN  = RAIN
  21          END,
  22    UNTIL RAIN = 99999,
```

Figure 5: Program Requiring Shallow Bug Reasoning

Although the Pascal portions of these plans match fairly closely, difficulties arise when the subgoals are matched against the program. Consider first the REPEAT loop plan. It indicates that there should be an *Input* subgoal at the top of the loop, and the remainder of the loop should be enclosed in a *Sentinel Guard*, *i.e.*, a test to see if the sentinel value has been read. There is in fact a READ statement at line 6 which could satisfy the *Input* goal. However, the code which follows, at line 7, is not a sentinel guard; instead, it is a loop that performs more input. This indicates that there is a flaw in this model of the code. A similar problem arises when PROUST tries to match the WHILE loop plan. The problem there is that the plan indicates that there should be an *Input* goal above the loop, but PROUST finds the loop at line 7 interposed between the initial read and the apparent main loop at line 13.

Mismatches between plans and codes are called plan differences; whenever a plan fails to match exactly a plan difference description is constructed describing the mismatches. There are two mechanisms which are used for resolving plan differences. One is to look for some other way of structuring the subgoals of the plans to match the code better. The other is to come up with an explanation of the plan mismatch in terms of bugs or plan transformations. Both mechanisms are needed in this example.

The first step in resolving the plan differences associated with the looping plans is to restructure the subgoals in order to reduce the differences. Besides the READ SINGLE VALUE plans, PROUST has other plans which can be used for input. One plan is a WHILE loop which tests the input for validity as it is being read, and rereads it if the data is not valid. This plan satisfies two goals simultaneously, *Input* and *Input Validation*. However, *Input Validation* is also on the goal agenda, so PROUST combines the two goals and matches the plan. The result is that in the case of the SENTINEL-CONTROLLED PROCESS-READ WHILE PLAN lines 6 through 12 is viewed as performing the initial input, and in the case of the REPEAT plan these same lines of code are viewed as the main input inside the loop.

Given these interpretations of the subgoals, the main loop plans still do not quite match. The remaining differences are compared against PROUST's bug catalog. This catalog has been built via empirical analyses of the bugs in hundreds of novice programs [9]. It consists of production rules which are triggered by plan differences and which chain, if necessary, in order to account for all the plan differences. In the case of the REPEAT loop plan, the plan difference is that a WHILE statement is found instead of an IF statement; this is listed in the bug catalog, along with the probable associated misconception. In the case of the WHILE loop plan, there are two plan differences: the *Input* subgoal is missing from inside the loop, and the entire loop is enclosed inside of another loop. The missing input is listed in the bug catalog; novices sometimes have the misconception that a READ statement is unnecessary in the loop if there is a READ statement elsewhere in the program. The enclosing loop bug is not listed in the catalog; that does not mean that this is an impossible bug, only that there is no canned explanation for this kind of plan difference.

We have thus constructed two different interpretations of the implementation of the *Sentinel-Controlled Input Sequence* in this program. There are others which we have not listed here. It is necessary to construct all these different interpretations, instead of just picking the first one that appears reasonable, because there is no absolute criterion for when a plan should be considered to match buggy code. The only way to interpret code when bugs are present is to consider the possible interpretations and pick one which appears to be better than the others. In other words, PROUST must perform differential diagnosis in order to pick the right interpretation. The intention model makes it possible to construct such a differential; PROUST uses it to predict possible plans and subgoal structures which might be present, thus enumerating hypotheses to consider.

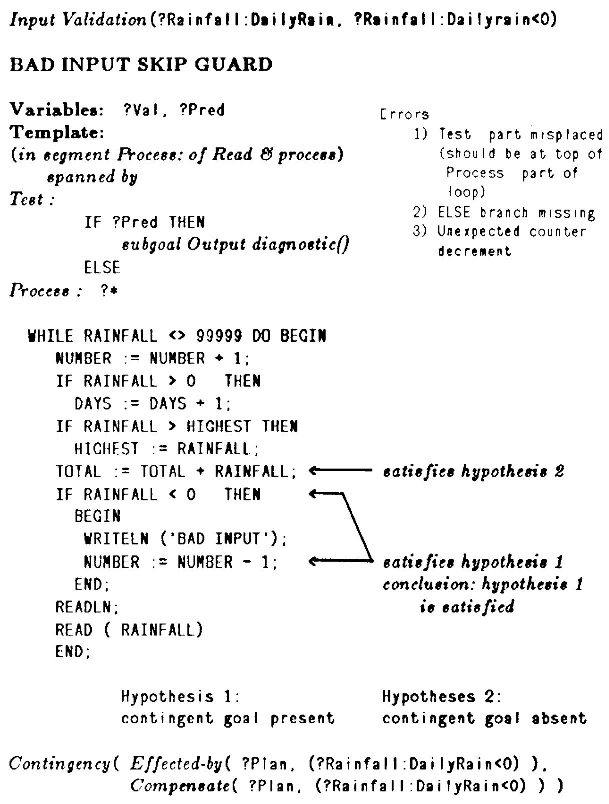
Choosing from among the possible interpretations proceeds as follows. If there is an interpretation which is reasonably complete, and which is superior to competing interpretations, PROUST picks it, and saves the alternatives, in case evidence comes up later which might invalidate the decision. Here the REPEAT loop interpretation is superior, because each part of the plan has been accounted for, albeit with bugs. The WHILE loop interpretation is not as good, because the embedded loop plan difference is unexplained, and because the *Input* subgoal inside the loop was never found. PROUST therefore adopts the REPEAT loop interpretation, and adds the "while-for-if" bug to the current diagnosis for this program.

We see in this example that although shallow reasoning is used to recognize plans and bugs, this works only because causal reasoning about the programmer's intentions provides a framework for performing plan and bug recognition, and for interpreting the results. The intention model makes it possible to employ differential diagnosis techniques, which helps PROUST arrive at the correct interpretation of the program even when bugs make it difficult to determine what plans the programmer was trying to use. In contrast, analysis methods which analyze the program behavior would probably be fooled by this program, because they would treat the WHILE statement at line 13 as a loop, rather than as an IF statement. Such a system might be able to determine that the program goes into an infinite loop, but it would not be able to explain to the programmer why his/her intentions were not realized.

4.3. Differentiating intention models

Figure 6 shows a program which requires deeper analysis of the programmer's intentions. This example illustrates how programming goals are sometimes realized indirectly in a program, by interacting with the implementation of other goals. Debugging such programs requires the ability to reason about goal interactions in order to differentiate models of the programmer's intentions. Causal reasoning about intentions is essential in this enterprise.

Let us examine how PROUST maps the goal *Input Validation* onto this program, *i.e.*, how it determines how bad input is filtered from the input stream. One plan which implements this goal is the BAD INPUT SKIP GUARD, which encloses the computations in the loop with an IF-THEN-ELSE statement which tests for bad input. PROUST discovers plan differences when it tries to match this plan against the program. PROUST can find a test for bad input in the loop, but it is too far down in the loop, and it does not have an ELSE branch. It also contains an unexpected counter decrement statement, NUMBER := NUMBER-1. It turns out that these plan differences are explained not by postulating a bug in the BAD INPUT SKIP GUARD plan, but by inferring an altogether different goal structure for the program.



```
Input Validation(?Rainfall:DailyRain, ?Rainfall:Dailyrain<0)

BAD INPUT SKIP GUARD

Variables:  ?Val, ?Pred                          Errors
Template:                                          1) Test  part misplaced
(in segment Process: of Read & process)               (should be at top of
    spanned by                                         Process  part of
Test :                                                 loop)
      IF ?Pred THEN                                2) ELSE branch missing
          subgoal Output diagnostic()              3) Unexpected counter
      ELSE                                             decrement
Process :  ?*

  WHILE RAINFALL <> 99999 DO BEGIN
     NUMBER := NUMBER + 1;
     IF RAINFALL > 0   THEN
       DAYS := DAYS + 1;
     IF RAINFALL > HIGHEST THEN
       HIGHEST := RAINFALL;
     TOTAL := TOTAL + RAINFALL;  <------  satisfies hypothesis 2
     IF RAINFALL < 0   THEN      <--\
       BEGIN                         \
        WRITELN ('BAD INPUT');        \
        NUMBER := NUMBER - 1;    <-----\  satisfies hypothesis 1
       END;                              conclusion: hypothesis 1
     READLN;                                 is satisfied
     READ ( RAINFALL)
     END;

         Hypothesis 1:                Hypotheses 2:
         contingent goal present      contingent goal absent

Contingency( Effected-by( ?Plan, (?Rainfall:DailyRain<0) ),
             Compensate( ?Plan, (?Rainfall:DailyRain<0) ) )
```

Figure 6: Differentiating Intention Models

PROUST assumed that a single plan would be used to implement the *Input Validation* goal. Instead, two plans are used in this program: one prints out a message when bad input is read in, and the other decrements the counter NUMBER when bad input is read in. In fact, for this design to be correct, there would have to be a ***third*** plan, which subtracts bad input from the running total, TOTAL. We must therefore reformulate the *Input Validation* as a ***contingent goal***: *Contingency*(*Effected-by*(?Plan, (?Rainfall:DailyRain<0)), *Compensate*(?Plan, (?Rainfall:DailyRain<0))). This goal states that whenever a plan is effected by the rainfall variable being less than zero, a goal of compensating for this effect must be added to the goal agenda. If we assume that bad input was being filtered in a contingent fashion, then input will be tested when it might effect the result; it would not be tested when the maximum, HIGHEST, is computed, for example. This hypothesis is generated by a bug rule which fires when PROUST tries to explain the plan differences listed in the previous paragraph. This rule stipulates that whenever a guard plan only guards part of the code that is supposed to guard, a new goal structure should be created in which the guard goal is reformulated as a contingent goal.

Testing a contingent goal hypothesis is difficult, because it depends upon what plans are used to implement the other goals in the agenda, and PROUST does not yet know what those plans are. PROUST must construct a differential of two goal structure hypotheses: hypothesis 1 holds that *Input Validation* has been reformulated as a contingent goal, and hypothesis 2 holds that the programmer neglected *Input Validation* altogether, and the code which appears to guard against bad input really serves another purpose. In order to test these hypotheses, PROUST activates two demons. The first demon tests hypothesis 1, by looking for plans which satisfy the contingency test and checking to see that bad input has been accounted for. The other demon tests hypothesis 2, by looking for cases where bad input should have been checked for but was not, and for alternative explanations for the code which is attributed to the contingent goal. Each demon finds one case supporting its respective hypothesis. The program compensates for the effect of bad input on the counter, NUMBER, which serves as evidence for the contingent goal hypothesis. However, the program does not compensate for the effect of bad input on the running total, TOTAL; this serves as evidence that the *Input Validation* goal was not implemented at all. This does not mean that the hypotheses are equally good, however. Hypothesis 1 can account for the running total being unguarded if we presume that the programmer left that case out by mistake. Hypothesis 2 cannot account for NUMBER := NUMBER-1 line at all; it would have to be dismissed as spurious code. PROUST avoids program interpretations which cannot account for portions of the code. Therefore PROUST can assume that the programmer has the contingent input validation goal in mind.

Knowledge about how a goal structure was derived by the student makes it possible for PROUST to help the student improve his programming style. This program can be fixed by adding a line TOTAL := TOTAL-RAINFALL next to the NUMBER := NUMBER-1 line. This is not the right correction to suggest: it was a mistake for the programmer to validate the input in a contingent fashion in the first place. A single plan could have implemented the input validation directly, and the fact that the programmer overlooked one of the contingencies demonstrates that indirect goal implementations are harder to perform correctly. As we see in the output which PROUST generates for this bug, shown in Figure 7, PROUST suggests to the student that he re-implement the *Input Validation* goal. Thus causal reasoning about intentions not only makes it possible to find bugs in complex programs, it makes it possible to correct the reasoning that led to the occurrence of the bugs.

```
The sum is not shielded against invalid input. If the user
types in a bad value it will be included in the sum. I noticed
that you do test for bad input elsewhere. Your program
would really be simpler if it tested the input in one place,
before it is used, so bugs like this would not crop up.
```

Figure 7: PROUST output for the program in Figure 6

5. Results

To test PROUST, we performed off-line analysis of the first syntactically correct versions of 206 different solutions to the Rainfall Problem. A team of graders debugged the same programs by hand, to determine the actual number of bugs present. The results are shown in Figure 8, labeled "Test 1". For each program PROUST generates one of three kinds of analyses.

- *Complete analysis*: the mapping between goal structure and code is complete enough that PROUST regards it to be fully reliable.
- *Partial analysis*: significant portions of the program were understood, but parts of the program could not be analyzed. PROUST deletes from its bug report any bugs whose analysis might be effected by the unanalyzable code.

- *No analysis*: PROUST's analysis of the program was very fragmentary, and unreliable, and was therefore not presented to the student.

In Test 1, 75% of the programs received complete analyses; PROUST found 95% of the bugs in these programs, including many that the graders missed. 20% of the programs were partially analyzed, and 5% got no analysis.

	Test 1	Test 2	Test 2 Repeated
Total number of programs:	206	76	76
Number analyzed completely:	155 (75%)	30 (39%)	53 (70%)
Total number of bugs:	531	133	252
Bugs recognized correctly:	502 (95%)	131 (98%)	247 (98%)
Bugs not recognized:	29 (5%)	2 (2%)	5 (2%)
False alarms:	46	5	18
Number analyzed partially:	40 (20%)	33 (43%)	19 (25%)
Total number of bugs:	220	163	105
Bugs recognized correctly:	79 (36%)	65 (40%)	42 (40%)
Bugs deleted from bug report:	80 (36%)	58 (36%)	36 (34%)
Bugs not recognized:	61 (28%)	40 (25%)	27 (26%)
False alarms:	36	20	17
Number unanalyzed:	11 (5%)	13 (17%)	4 (5%)

Figure 8: Results of running PROUST

We have recently made on-line tests of PROUST in an introductory programming course. The column labeled "Test 2" summarizes PROUST's performance. Unfortunately the percentage of complete analyses went down. This turned out to be because of problems in transporting PROUST from the research environment to the classroom environment, and were not due to essential flaws in PROUST itself. We corrected these problems and re-ran PROUST on the same set of data; this time PROUST's performance was comparable to what it was in Test 1. We also ran PROUST on another programming problem; the results of that test have yet to be tabulated.

6. Concluding Remarks

We have argued that intention-based understanding is needed in order to diagnose errors effectively in novice programs. Knowledge of intentions makes it possible for PROUST to grapple with the high degree of variability in novice programs and novice programming errors, and achieve a high level of performance. Intention-based diagnosis is complex, but our results suggest that it is tractable for non-trivial programs. This gives us optimism that the remaining obstacles to achieving high performance over a wide range of student populations and programming problems can be overcome in due course.

References

1. Adam, A. and Laurent, J. "LAURA, A System to Debug Student Programs." *Artificial Intelligence 15* (1980), 75-122.

2. Bonar, J. and Soloway, E. Uncovering Principles of Novice Programming. SIGPLAN-SIGACT Tenth Symposium on the Principles of Programming Languages, 1983.

3. Chandrasekaran, B. and Mittal, S. Deep Versus Compiled Knowledge Approaches to Diagnostic Problem-Solving. Proc. of the Nat. Conf. on Artificial Intelligence, AAAI, August, 1982, pp. 349-354.

4. Davis, R. Diagnosis via Causal Reasoning: Paths of Interaction and the Locality Principle. Proc. of the Nat. Conf. on Artifical Intelligence, AAAI, August, 1983, pp. 88-94.

5. Genesereth, M. Diagnosis Using Hierarchical Design Models. Proc. of the Nat. Conf. on Art. Intelligence, 1982, pp. 278-283.

6. Harandi, M.T. Knowledge-Based Program Debugging: a Heuristic Model. Proceedings of the 1983 SOFTFAIR, SoftFair, 1983.

7. Johnson, W.L. Intention-Based Diagnosis of Programming Errors. Tech. Rept. forthcoming, Yale University Department of Computer Sci., 1984.

8. Johnson, L., Draper, S., and Soloway, E. Classifying Bugs is a Tricky Business. Proc. NASA Workshop on Soft. Eng., 1983.

9. Johnson, W.L., Soloway, E., Cutler, B., and Draper, S. Bug Collection: I. Tech. Rept. 296, Dept. of Computer Science, Yale University, October, 1983.

10. Lukey, F.J. "Understanding and Debugging Programs." *Int. J. of Man-Machine Studies 12* (1980), 189-202.

11. Pople, H. E. Heuristic Methods for Imposing Structure on Ill Structured Problems: The Structuring of Medical Diagnostics. In Szolovits, P., Ed., *Artificial Intelligence in Medicine*, West View Press, 1982.

12. Sedlmeyer, R. L. and Johnson, P. E. Diagnostic Reasoning in Software Fault Localization. Proceedings of the SIGSOFT Workshop on High-Level Debugging, SIGSOFT, Asilomar, Calif., 1983.

13. Shapiro, D. G. Sniffer: a System that Understands Bugs. Tech. Rept. AI Memo 638, MIT Artificial Intelligence Laboratory, June, 1981.

14. Shapiro, E.. *Algorithmic Program Debugging.* MIT Press, Cambridge, Mass., 1982.

15. Shortliffe, E.H.. *Computer-Based Medical Consultations: MYCIN.* American Elsevier Publishing Co., New York, 1976.

16. Soloway, E., Rubin, E., Woolf, B., Bonar, J., and Johnson, W. L. "MENO-II: An AI-Based Programming Tutor." *Journal of Computer-Based Instruction 10*, 1 (1983).

17. Soloway, E. and Ehrlich, K. "Empirical Investigations of Programming Knowledge." *IEEE Transactions of Software Engineering SE-10*, In press (1984).