From: AAAI-86 Proceedings. Copyright ©1986, AAAI (www.aaai.org). All rights reserved.

An Analysis of Tutorial Reasoning About Programming Bugs

David C. Littman, Jeannine Pinto & Elliot Soloway
Cognition and Programming Project
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

A significant portion of tutorial interactions revolve around the bugs a student makes. When a tutor performs an intervention to help a student fix a programming bug, the problem of deciding *which* intervention to perform requires extensive reasoning. In this paper, we identify five tutorial considerations tutors appear to use when they reason about how to construct tutorial interventions for students' bugs. Using data collected from human tutors working in the domain of introductory computer programming, we identify the knowledge tutors use when they reason about the five considerations and show that tutors are *consistent* in the ways that they use the kinds of knowledge to reason about students' bugs. In this paper we illustrate our findings of tutorial consistency by showing that tutors are consistent in how they reason about bug criticality and bug categories. We suggest some implications of these empirical findings for the construction of intelligent tutoring systems.

1 Introduction: The Problem of Tutorial Consistency

A key issue for designers of Intelligent Tutoring Systems is how to treat students' bugs. Both the research of others (e.g., Collins and Stevens (1976)) and our own work (Littman, Pinto, and Soloway (1985)) suggest that bugs play a central role in tutoring. In a sense, tutors use bugs to *drive* the tutorial process: bugs help the tutor understand *what the student does not understand* and they provide a ready forum for communication with students since all students want to fix their bugs. Though most tutors try to help students fix bugs, the skill of expert tutors, and therefore effective Intelligent Tutoring Systems, lies in *how they use bugs* in their tutorial interventions. A simple first-order model for using bugs in tutoring would have three steps:

- identify the bug
- look up an appropriate response to the bug in a database of tutorial responses
- deliver the appropriate response to the student.

This three step model of tutorial intervention, which is essentially the model used by CAI systems (Carbonell (1970)), does not require the tutoring system to *reason* either about what knowledge to teach a student who makes a bug nor about how to teach the knowledge. In a sense, all the tutorial knowledge possessed by such systems is "compiled". The three step model may be appropriate for tutoring students when their bugs do not reflect deep misunderstandings, or when one bug always should get the same intervention. However, it seems unlikely to be effective in domains such as computer programming where students' bugs are often related and may reflect deep misconceptions about how to solve problems or about the constructs of the programming language. In complex domains such as programming, tutors seem to engage in extensive reasoning about how to tutor students who make serious bugs. As an example of the kinds of issues a tutor reasons about when tutoring students in complex domains, consider the ostensibly simple problem of *when* to deliver tutorial interventions. Two opposite strategies been proposed:

- The LISP tutor of John Anderson's group (cf. Anderson, Boyle, Farrell, and Reiser (1984)) provides *immediate feedback to the student* on *all* bugs the student makes.
- The WEST tutor of Brown and Burton (1982) plays a *very* conservative "coaching" role with the goal of minimizing interruptions of the student's problem solving. WEST takes a "wait-and-see" attitude to interrupting the student, trying to collect diagnostic information from *patterns of bugs*.

Since the three step model seems inappropriate for the Intelligent Tutoring Systems that will have to be built for complex domains, and since there appears to be considerable controversy about the generation of tutorial interventions, we decided that it would be useful to study human tutors in an effort to determine how they reason about tutorial interventions for students who make bugs.

Our general approach to studying how tutors reason about bugs was to identify several issues that we believe tutors reason about to generate their interventions. From interviews with tutors, and videotapes of interactive tutoring sessions, we identified five main issues tutors reason about when they generate their tutorial interventions. Each of these issues, called a *tutorial consideration*, influences the tutor's decisions about which bugs to tutor, when to tutor them, and how to tutor them. The five tutorial considerations are:

1. How *critical* the bugs are
2. What *category* the bugs fall into
3. What *caused* the bugs
4. What tutorial *goals* are appropriate for tutoring the bugs

The research reported in this paper was cosponsored by the Personnel and Training Division 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-k0714, Contract Authority Identification Number 154-492.

5. What tutorial *interventions* would achieve the tutorial goals

With this very general set of tutorial considerations in mind we designed a variant of a protocol study (Newell and Simon (1972)) that was intended to present tutors with situations that would lead them to reason about the five tutorial considerations. Tutors were presented with buggy programs actually written by students in an introductory PASCAL programming class and asked to answer questions designed to elicit reasoning about the five tutorial considerations. For example, we asked tutors why they thought the student who wrote each program made the bugs, what goals they had for tutoring the student, and what they would actually do to tutor the student.

During our initial analysis of the data set, we have had the goal of simply identifying and describing the kinds of knowledge tutors have and the factors that tutors take into account when they reasoned about the five tutorial considerations. By abstracting the responses of many tutors to the same questions, we have begun to identify different kinds of knowledge tutors use and the factors that they weigh when they make decisions about the tutorial considerations. Our initial description of the data, therefore, is in terms of:

- tutorial considerations
- kinds of knowledge tutors use in reasoning about tutorial considerations
- factors that comprise the knowledge tutors use

Though we do not yet have a computer program that implements our findings about human tutors, we definitely plan to use the information we acquire from this study to guide our development of Intelligent Tutoring Systems. Since at this point we are trying to develop a descriptive vocabulary that permits us to express tutorial knowledge and reasoning, and to describe such knowledge and reasoning, our current research is more appropriately viewed as theory ***building*** than as theory *application*. Hence, in this paper we present part of the vocabulary and use it to show that tutors are consistent when reasoning about tutoring students' bugs.

One of the major concerns of our research has been the problem of consistency of tutorial reasoning. Because tutors use so many kinds of information to decide how to tutor a student's bug, it seems plausible to hypothesize that different tutors would be inconsistent in the ways they reason about either identical bugs or different bugs. The problem of tutorial consistency is important to designers of Intelligent Tutoring Systems since, if human tutors were entirely inconsistent in their generation of tutorial interventions, using human tutors as models for machine tutors would not be useful. Absence of tutorial consistency would imply that there is no reason to prefer any one method of generating tutorial interventions over any other method on the grounds that human tutors find one method especially effective. Fortunately, there are at least two sources of evidence for tutorial consistency. First, Collins and Stevens (1976), in a study of "super-teachers", identified several Socratic tutorial strategies that their teachers used; many of the strategies identified by Collins and Stevens (1976) found their way into the Socratic WHY tutor (Stevens, Collins, and Goldin (1982)), Woolf's programming tutor for students in introductory programming courses (Woolf (1985)), and Clancey's GUIDON program for teaching the skill of medical diagnosis (Clancey (1983)). Second, our analyses of the data we gathered from human tutors suggest that tutors are consistent in the ways in which they reason about how to tutor students who make bugs.

This paper is organized as follows:

- First, in Section 2, we describe the experiment we conducted to collect data about tutorial consistency.
- Second, in Section 3, we present an example which illustrates how two tutors reason in the same way about the same bug.
- Third, in Sections 4 and 5, we describe bug criticality and bug categories and present statistical evidence that tutors are consistent in reasoning about both.
- Finally, in Section 6, we draw some conclusions and implications of our study.

Though we do not present analyses of all of the five considerations tutors take into account in deciding how to tutor students' bugs, the analyses of bug criticality and bug categories illustrate our general findings which apply equally to bug categories, the causes of bugs, tutorial goals, and tutorial interventions. A complete analysis of the consistency of all five types of knowledge is presented in Littman, Pinto, and Soloway (1986).

2 Methods

2.1 Subjects

Eleven Yale University graduate and advanced undergraduate students participated in this study. Each had extensive tutoring experience. The range in tutorial experience was from 150 to over 2000 hours. Each subject could program competently in PASCAL as well as in a variety of other languages.

2.2 Task

Subjects received five buggy programs actually written by introductory programming students along with the same questionnaire about each program. The programs were written in response to the *Rainfall Assignment*, which was assigned during the fifth week of class. The assignment is shown in Figure 1 and a program that correctly solves the assignment is shown in Figure 2. To reproduce the typical situation a programming tutor faced in introductory PASCAL programming courses, the buggy programs contained an average of 6 bugs. For each student's program, tutors were asked to imagine themselves tutoring the student who wrote the program and to answer each of the questions in the questionnaire. The questionnaires were displayed side-by-side with the buggy programs on an Apollo DN300 multi-window workstation. Subjects typed their answers to each question, pressed a pre-assigned key to go to the next question, and continued until they were finished. Subjects were allowed to work at their own pace. Most subjects needed at least four hours to complete all the questionnaires.

The questionnaire was designed to prompt the tutors for their thoughts as they considered how they would tutor the student who wrote the program. For example, subjects decided whether a bug would be tutored alone or in a group with other bugs. They also indicated the order in which they would tutor the groups of bugs as well the goals they had for tutoring each bug and the methods they would use to achieve the goals.

While we realize that our experimental design presented subjects with a somewhat artificial situation, we were very encouraged by how engaging our subjects found the task. Subjects took the task seriously, spending as much as 15 hours to complete it. Informal debriefing interviews further convinced us that the tutors felt their responses were valid and would have been essentially the same in a real tutoring session.

The Noah Problem: Noah needs to keep track of the rainfall in the New Haven area to determine when to launch his ark. Write a program so he can do this. Your program should read the rainfall for each day, stopping when Noah type "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 1: The Rainfall Assignment

```
Program Rainfall(input,output);
 Var DailyRainfall,TotalRainfall,MaxRainfall,Average : Real;
   RainyDays,TotalDays : Integer;
 Begin
  RainyDays:= 0; TotalDays:= 0;
  MaxRainfall:= 0; TotalRainfall:= 0;
  Writeln ('Please Enter Amount of Rainfall');
  Readln(DailyRainfall);
  While (DailyRainfall <> 99999) Do
   Begin
     If DailyRainfall >= 0 Then
      Begin
        If DailyRainfall > 0 Then
          RainyDays := RainyDays + 1;
        TotalRainfall := TotalRainfall + DailyRainfall;
        If DailyRainfall > MaxRainfall Then
          MaxRainfall := DailyRainfall;
        TotalDays := TotalDays + 1
      End;
     Else Writeln ('Rainfall Must Be Greater Than 0');
    Read(DailyRainfall)
  End;
  If TotalDaysCounter > 0 Then Begin
     Average := TotalRainfall/TotalDays;
     Writeln('Average is: ', Average: 0:2);
     Writeln('Maximum is: ', MaxRainfall: 0:2);
     Writeln('Total Number of Days is: ', TotalDays);
     Writeln('Total Number of Rainy Days is: ', RainyDays)
  End;
  Else Writeln('No Valid Days Entered.');
 End.
```

Figure 2: Sample Correct Rainfall Program

2.3 Choice of Bugs for Analysis

For this paper, we analyzed 16 of the 36 bugs in the five programs. The 16 bugs represent the range of bugs in the experiment. Criteria for including bugs in the analyses were:

- Each bug represented a type of bug tutors frequently encounter.
- No more than one of each type of bug was included *unless* the same bug appeared in two very different contexts.
- Both *mundane* bugs and *interesting* bugs were chosen. An example of a mundane bug is failing to include an initialization of a counter variable. An example of an interesting bug is employing a complex IF-THEN construct for what should be a simple update of a counter variable.
- Bugs were included that produce both obvious effects on the behavior of the program (e.g., a missing READLN of the loop-control variable) and bugs that produce subtle effects on the behavior of the program (e.g., initialization of a counter variable to one more than its correct initial value.)

2.4 Data Scoring and Reliability of Scoring

Each response of each tutor was evaluated to identify knowledge relevant to each of the five tutorial considerations. In this section we illustrate the scoring of protocols with an example of a tutor's criticality considerations; we also present the criteria for protocol scoring reliability.

Scoring the Data: We illustrate the scoring of the protocol data by showing 1) how Tutor **1**'s bug criticality rating is derived and 2) how we score the factors the tutor identified in reasoning about bug criticality. Figure 3 shows a bug made by a student who was attempting to solve the *Rainfall Assignment*. The student spuriously assigned 0 to the variable intended to contain the value of rainfall entered by the user *immediately after the user has entered a value for* DailyRainfall. Our analysis of each tutor's reasoning about bug criticality is in terms of two measures:

- The tutor's criticality rating assigned to the bug based on the tutor's statements and
- the bug criticality factors the tutor identified in reasoning about the bug.

Figure 4 shows the template used to score each tutor's reasoning about bug criticality. The template consists of two parts, a field for the Tutor's Overall Criticality Rating and a list of the factors associated with reasoning about bug criticality.[1] The following quotation shows the statements Tutor **1** made that are relevant to the bug criticality consideration.

> **Tutor 1:** "(1.) [This is a] trivial error ... that must be fixed to get good output. (2.) Simple mistake. (3.) Forgetting that Rainfall was losing its value .... "

The first part of Tutor **1**'s first sentence and the entire second sentence show that he does not believe that the spurious initialization bug is very critical. As shown in Figure 4, the tutor's response to the bug was coded as LOW CRITICAL, the lowest value on the three point scale we used to score tutors' evaluations of bug criticality. Sentence three shows that Tutor **1** does not believe a deep problem of the Student's Understanding was responsible for the bug; the student simply forgot. Thus, the scoring template contains an "X" in the column for Student's Understanding to show that the tutor identified this factor. Finally, in the second half of the first sentence the tutor says that the bug must be fixed to get good output. This identifies the factor of Program Behavior Preconditions since the bug must be fixed for the program to output correct values.

[1] A description of the meanings of each of the factors is presented in Figure 6 in Section 4.

```
Program Rainfall(input,output);
  ...
  TotalRainfall := 0;
  Writeln ('ENTER AMOUNT OF RAINFALL');
  Readln(DailyRainfall);
  DailyRainfall := 0; BUG: Assignment of 0 to DailyRainfall
                      Clobbers Initial Value
  While (DailyRainfall <> 99999) Do
   Begin
    ...
   End;
  ...
```

Figure 3: Bug: Assignment of 0 to DailyRainfall Clobbers Initial Value

Reliability of Scoring: The data we analyze in this paper are based on subjective interpretation of tutors' responses. They are not, for example, reaction times or numbers of errors. Rather, the statements tutors made in response to the questionnaires were *interpreted* in order to produce the data. To assess whether the data derived from the protocol statements accurately reflect the cognitive processes which generated them, such data are normally subjected to *reliability analysis*. If the interpretations of the protocol responses are sufficiently reliable, then they are judged to reflect cognitive processes of the subjects who produced them. Reliability of encodings of the protocol responses was assessed by two rules:

- If the coder of a response had any question about the correct label for the response, the response was jointly encoded by more than one coder.
- A response was eliminated from the analysis if it could not be encoded, or two or more coders disagreed on the appropriate encoding.

A random sample of approximately 30% of encodings of each kind of knowledge was evaluated by more than one coder. The random sampling of mutually evaluated responses resulted in less than five percent of the data being shifted from one encoding to another.

Tutor's Overall Criticality Rating LOW CRITICAL

FACTORS IDENTIFIED BY TUTOR

Name of Factor	Factors Present
Student's Understanding	X
Impact on the Tutorial Plan	
Knowledge Preconditions	
Program Behavior Preconditions	X
Bug Dependencies	
Student's Ability to Find and Fix Bug Alone	
Student's Motivation	
Diagnostic Opportunities	

Figure 4: Scoring Tutor 1's Bug Criticality Consideration

3 Tutorial Consistency: An Illustration

In this section, we present an example of two tutors reasoning about the same bug. Our intent is to illustrate for the reader the kind of data tutors generated in our study and to provide some intuitions about how we analyzed our protocol data.

3.1 Two Tutors Reason About the Same Bug

Figure 3 shows the spurious initialization bug we considered in Section 2. To illustrate similar reasoning of two tutors about the five tutorial considerations, we present and discuss quotations from their protocols as they reasoned about how to tutor the bug.

Tutorial Consideration 1: Bug Criticality

Neither Tutor **2** nor Tutor **3** felt that the bug shown in Figure 3 was very critical. The following quotations show why both tutors were coded as having the same bug criticality rating:

> **Tutor 2:** "It's a small but annoying and pervasive problem ..."
>
> **Tutor 3:** "... this does seem like a relatively trivial bug."

Even though the bug interferes seriously with the behavior of the student's program, neither tutor believed it is a "serious" bug; we will see why when we discuss the tutors' reasoning about the causes of the bug.

Tutorial Consideration 2: Bug Category

Both tutors believed that the student who made the bug failed to translate correctly the conceptual object for some variable into its correct name in the program. Instead of initializing the intended variable to 0, the failure to translate the conceptual object into its corresponding code caused the student to initialize the wrong variable, DailyRainfall. The following quotations were the basis of our encoding of the tutors' categorizations of the bug as a failure to translate correctly from conceptual objects to code:

> **Tutor 2:** "Syntactic similarity of the two variable names ..."
>
> **Tutor 3:** "Just mixed up variable names ..."

The reason the tutors believed the student made the bug identifies the category of bug: namely those bugs that arise from failures to translate conceptual objects correctly to the code that instantiates the conceptual objects.

Tutorial Consideration 3: Bug Cause

Tutor **2** and Tutor **3** identified essentially the same cause for the bug.

> **Tutor 2:** "... mixing up the purpose of the variables ..."
>
> **Tutor 3:** "I think the student was confusing TotalRainfall with DailyRainfall ..."

The tutors attributed the cause of the bug to the student's confusing the variable DailyRainfall with another, similarly named, variable. Evidently they felt that the student had correctly identified the conceptual purpose of the two variables, had given them appropriate names, and then confused the two names because they were so similar. We will see evidence for this view in the next quotations which illustrate the tutors' goals in tutoring the bug.

Tutorial Consideration 4: Tutorial Goals

Both tutors were interested in teaching the student to use variables names that prevent confusion when code. The

following quotations show that both tutors wanted to teach the student the variable-naming heuristic.

> **Tutor 2:** "I would explain that there seems to be a name confusion ..."

> **Tutor 3:** "Be careful that you name your variables distinctly enough so that you do not get confused about which role they are serving."

Notice that the tutorial goals identified by Tutor **2** and Tutor **3** are reasonable in light of their explanations of the cause of the bug.

Tutorial Consideration 5: Tutorial Interventions
The following quotations show that both tutors wanted to draw the student's attention to the mismatch between the goal the student had for the variable TotalRainfall and what actually happens to it.

> **Tutor 2:** "One could ask a leading "**WHY**" question ... asking him to justify his coding ..."

> **Tutor 3:** "I could ask them *if they meant to be initializing* TotalRainfall instead of DailyRainfall"

Both tutors selected the strategy of juxtaposing for the student the student's intentions, or goals, with the actual code in the program. This general kind of tutorial intervention was extremely popular with our tutors and appears to serve the purpose of forcing the student to identify conflicts between intentions and actions.[2]

The tutors' responses to the bug shown in Figure 3 illustrate how two tutors can have essentially the same "perspective" on the same bug. In the next section of the paper, we identify the factors that tutors take into account when they reason about bug criticality show, statistically, that tutors are consistent in the ways they reason about bug criticality.

4 Bug Criticality

In planning tutorial sessions, tutors make decisions about which bugs to focus on explicitly and which bugs to tutor only as opportunities arise. When our tutors identified bugs that they intended to focus on in their tutorial sessions, they gave reasons that made it clear that they felt that those bugs were more critical than others. As we analyzed tutors' responses to the buggy-program scenario questionnaires, we identified several factors that seemed to play a role in their decisions about which bugs to focus on. For example, tutors focused on bugs that might have been caused by serious misconceptions, bugs that suggested the student lacked important knowledge or skills, and bugs that interfered with the behavior of the program so much that the student would be unable to debug it.

In this section of the paper we describe the main factors that our tutors used to reason about bug criticality. As examples of critical and noncritical bugs, suppose a student writes a solution to the *Rainfall Assignment* in which the update for the variable containing the total amount of rainfall, TotalRain, is like the fragment of code labelled as **BUG 1** in Figure 5. Instead of simply updating the variable TotalRain by adding in the value of DailyRainfall, the student has written the update using a very strange, malformed, IF-THEN statement to "guard" the update. Virtually every one of the tutors in our study judged the malformed update bug to be very critical because the bug could be symptomatic of a deep misconception about how to update variables. On the other hand, most novice programmers leave output variables unguarded against the case of no valid input: **BUG 2** in Figure 5 is an unguarded output bug. Our tutors uniformly considered **BUG 2** to be uncritical because it does not suggest the student who wrote the program has any deep misunderstandings about programming. The student probably just forgot to test this case.

```
...
Writeln ('ENTER AMOUNT OF RAINFALL'),
Read(DailyRainfall),
While (DailyRainfall <> Sentinel) Do
  Begin
    Writeln('ENTER AMOUNT OF RAINFALL')
    ...
                                   BUG 1: Malformed Update
If TotalRain = Tot + DailyRainfall  Of TotalRain
   Then Tot := TotalRain;
    ...
  End,
...
                          BUG 2: Output of TotalRain
                            Unguarded on No Input
Writeln('The Total Rainfall is: ', TotalRain);
```

Figure 5: Critical Bug: Severely Malformed Update

Figure 6 shows the major factors and subfactors we used to score tutors' reasoning about bug criticality. Our analyses of the tutors' data revealed two major factors tutors take into account when reasoning about the bug criticality tutorial consideration:

- What the bug implies about **Student's Understanding**
- The bug's **Impact on the Tutorial Plan** of the bug.

The major factor of **Student's Understanding** includes knowledge the student should already have and knowledge the student should acquire by doing the current assignment. For example, one tutor was scored as using this factor when she said the following about a student who did not include a ReadIn statement in the loop to get the new value of DailyRainfall:

> "*The student doesn't understand* that the loop is driven by input and therefore must contain an instruction to get input."

The major factor of **Impact on the Tutorial Plan**, which is more complex than **Student's Understanding**, is comprised of six subfactors. We present quotations to illustrate two main subfactors.

- **Knowledge Preconditions:** *used to justify tutoring one bug after another bug*
  The following quotation shows the tutor reasoning that tutoring one bug was a necessary precondition to tutoring some other bugs.

  > "These [bugs] make sense to follow that [bug] ... *we can presume that now the student has a full understanding of* initialization [the problem tutored first.]"

[2]This strategy was identified by Collins and Stevens (1976) as a central technique of the "Socratic Method" and formed the basis of the tutorial strategies implemented in the WHY tutor.

- **Program Behavior Preconditions:** *used to justify tutoring a bug first*
  In this quotation the tutor says that he started with a particular bug because fixing that bug was necessary to get the program to run even reasonably well.

  > "It's important in terms of *getting the program to run in any form*, thus gets precedence over later bugs."

Further discussion of tutors' reasoning about factors that have an impact on the tutorial plan can be found in Littman, Pinto, and Soloway (1985).

4.1 Tutors' Agreement on Criticality of Bugs

In this section, we identify two major findings that illustrate consistency of tutorial reasoning about the bug criticality tutorial consideration.

- First, tutors assign consistent criticality ratings to bugs.
- Second, tutors agree on the factors and subfactors for why bugs are critical.

- **Student's Understanding:** What problem solving and programming concepts does the student know?
- **Impact on the Tutorial Plan:** How should the tutorial plan be formulated?
  - **Knowledge Preconditions:** Knowledge the student must have to learn key material that the tutor intends to teach during the tutorial session.
  - **Program Behavior Preconditions:** Does the program's current behavior obstruct the tutor's plan for tutoring a bug the tutor wants to address?
  - **Bug Dependencies:** Bugs that, together, interact to produce program behavior.
  - **Student's Ability to Find and Fix Bug Alone:** The student's ability handle a bug without the tutor's assistance.
  - **Student's Motivation:** The tutor's assessment of whether the student needs to be handled with "kid gloves".
  - **Diagnostic Opportunities:** Would addressing this bug provide the tutor with useful information about the student's programming knowledge and programming skills?

Figure 6: Factors Affecting Bug Criticality

If tutors did not agree on the criticality of bugs, then the search for consistency of reasoning about bug criticality would be compromised. Our analyses show that tutors agreed very strongly about which of the 16 bugs were high critical, which bugs were medium critical, and which bugs were low critical. A Chi-squared analysis showed statistical significance of consistency of tutors reasoning about bug criticality ($\chi^2 = 119.6$, df $= 30$, $p < .01$).

It is possible that tutors would agree about bug criticality, yet would not identify the same factors and subfactors in their reasoning. Our data, however, show that tutors agree on the factors and subfactors, as shown in Figure 6, for why a particular bug is of high, medium, or low criticality. Chi-squared analysis of tutors' consistency in identifying particular factors and subfactors associated with particular bugs was statistically significant ($\chi^2 = 209$, df $= 140$, $p < .01$).

In summary, we have found that tutors see some bugs as being more critical than others. In addition, statistical analyses of their reasoning about bug criticality show that tutors are consistent both with respect to the criticality of bugs and the factors and subfactors that are associated with the criticality of bugs.

5 Bug Categories

When students attempt to solve the *Rainfall Assignment*, their first syntactically correct programs contain approximately six bugs each (Johnson, Soloway, Cutler, and Draper, 1983.) Instead of reasoning about each bug individually, tutors appear to use knowledge about *kinds* of bugs to help them determine both why the student made the bug and what to do to help the student. For example, if a student solving the *Rainfall Assignment* does not protect the calculation of the average against division by zero and also neglects to protect the output of the average against the case of no input data, a tutor might categorize both bugs as "missing boundary guards". Tutors appear to categorize bugs according to a coarse model of the program generation process and make a gross distinction between bugs that arise during program generation and bugs that arise during program verification; furthermore, they break the program generation phase into three subphases.[3] We now identify the three subphases of the generation category and present a quotation for each that shows the sort of statement that would be scored as referring to the subphase.

- **Decomposition:** *Figuring out what to do to solve problem.*
  In the following quotation the tutor shows he thinks the student failed to decompose correctly the problem of getting values of the rainfall variable into the two components of getting an initial value and getting each new value in the loop.

  > "I think the student knew they had read in `DailyRainfall` once and *thought that would be enough.*"

- **Mapping:** *Translating one level of problem analysis into another level (e.g., translating problem goals into plans to achieve the goals.)*
  The next quotation illustrates a tutor responding to a student who failed to protect the accumulator for `TotalRainfall` against adding in the sentinel value, 99999. To compensate for adding in 99999, the student subtracted 99999 from `TotalRainfall` just before calculating `AverageRainfall`.

  > "The student *plans to add in the sentinel (99999) and then remove it later.* I think this is very bad."

- **Composition:** *Coordinating solutions for different goals.*
  This quotation shows that the tutor believed the student failed to *compose* the main loop correctly with other actions the student wanted the

[3]While tutors did identify some subphases of the Verification category, subcategories of Verification were not stable and so we do not report them here.

program to take. The student's bug was to place below the loop the update of the variable accumulating the total amount of rainfall.

> "Common problem — Things outside the loop *which should be inside* [the loop.]"

5.1 Tutors' Agreement on Bug Categorization

In this section, we present two main findings for bug categorization:

- First, tutors agree in categorizing bugs as **Generation** or **Verification** bugs.
- Second, tutors agree in categorizing bugs as either **Decomposition**, **Mapping**, or **Composition** bugs.

Tutors were consistent in their categorization of bugs as arising during program generation or program verification, which constitutes the coarsest distinction of the bug category system. The consistency of tutors' categorizations of bugs as generation or verification bugs is demonstrated by the statistically significant Chi-square value for the test ($\chi^2 = 25.9$, df $= 15$, $p < .05$).

The major category, program **Generation**, is composed of three subphases: **Decomposition**, the attempt to determine how to solve the problem, **Mapping**, translating one level of problem solution into a more concrete level, and **Composition**, recombining the parts of a problem solution. The statistically significant Chi-square value shows that tutors agreed in categorizing bugs in these three phases of program generation ($\chi^2 = 162.3$, df $= 30$, $p < .01$).

In summary, tutors appear to categorize bugs into groups according to a coarse model of program generation and verification. When tutors' statements about bugs are analyzed to see how they categorized the bugs, we find that tutors consistently describe bugs in terms of a coarse model of program generation.

6 Conclusions, Implications and Future Directions

In this paper we have identified five tutorial considerations that tutors take into account when they reason about how to tutor students' bugs and we have provided vocabulary that permits us to describe and analyze patterns in tutors' reasoning about bug criticality and bug categories. In addition, we have presented statistical analyses that show that tutors are consistent in how they reason about bug criticality and bug categories.

There are two main implications of our ability to identify and describe consistency of tutorial reasoning. First, our data showing consistency of reasoning about individual considerations, such as bug criticality, suggest that we will be able to identify and describe consistent *patterns* of tutorial reasoning that *coordinate several tutorial considerations.* If we can identify and describe such patterns then we can *test their educational effectiveness* by selectively including various combinations of patterns into the same basic ITS, providing tutorial intervention to students with the modified ITS's, and empirically evaluating the effectiveness of the modified ITS's. Since intervention would be given to all students by the same basic ITS, differences in performance would be attributable to the specific tutorial patterns included in the ITS.

Second, when we have identified tutorial patterns that are educationally effective, we can build ITS's which incorporate them and avoid ineffective patterns. We will then be in a position to provide the same high quality tutorial experiences to every student who has access to a computer.

Our plans for the immediate future focus on identifying the patterns of tutorial reasoning that are educationally effective and building an Intelligent Tutoring System for programming which makes use of them. Our long range plans are directed toward empirically evaluating the effectiveness of the ITS for programming and using the tutorial principles we discover from our studies of human tutors to build ITS's for other domains.

7 References

Anderson, J., Boyle, C., Farrell, R., and Reiser, B. Cognitive principles in the design of computer tutors. Technical Report, Advanced Computer Tutoring Project, Carnegie-Mellon University, 1984.

R. Burton and Brown, J.S. An investigation of computer coaching for informal learning activities. In Intelligent tutoring systems. D. Sleeman and J. S. Brown (eds.), Academic Press, London, 1982.

Carbonell, J. AI in CAI: An artificial intelligence approach to computer assisted instruction. IEEE Transactions on Man-Machine Systems, MMS-M, 4, 1970.

Clancey, W. Guidon. Journal of computer-based instruction, Summer 1983, Vol. 10, Nos. 1 & 2, 8 - 15.

Collins, A. and Stevens, A. Goals and strategies of interactive teachers. Technical Report #3518, 1976 Bolt, Beranek, and Newman, Cambridge, MA.

Johnson, L., Soloway, E., Cutler, B., and Draper, S. Bug catalogue: I. Technical Report #286, 1983, Department of Computer Science, Yale University, New Haven CT.

Littman, D., Pinto, J., Soloway, E. Observations on tutorial expertise. Proceedings of IEEE Conference on Expert Systems in Government, Washington, D.C. 1985.

Littman, D., Pinto, J., Soloway, E. Consistency of tutorial reasoning. In preparation.

Newell, A. and Simon, H. Human problem solving. Prentice-Hall Englewood Cliffs, NJ, 1972.

Spohrer, J. and Soloway, E. Analyzing the high-frequency bugs in novice programs. To appear in: Workshop on empirical studies of programmers, E. Soloway and S. Iyengar (eds.), Ablex, Inc., 1986.

Stevens, A., Collins, A., and Goldin, S. Misconceptions in students' understanding. In Intelligent tutoring systems. D. Sleeman and J. S. Brown (eds.), Academic Press, London, 1982.

Woolf, B. Context dependent planning in a machine tutor. Doctoral Dissertation, University of Massachusetts, Amherst, MA 1984.