HUMAN-COMPUTER INTERACTION, 1985, Volume 1, pp. 133-161

Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers

Jeffrey Bonar
University of Pittsburgh

Elliot Soloway
Yale University

ABSTRACT

We present a process model to explain bugs produced by novices early in a programming course. The model was motivated by interviews with novice programmers solving simple programming problems. Our key idea is that many programming bugs can be explained by novices inappropriately using their knowledge of step-by-step procedural specifications in natural language. We view programming bugs as patches generated in response to an impasse reached by the novice while developing a program. We call such patching strategies *bug generators*. Several of our bug generators describe how natural language preprogramming knowledge is used by novices to create patches. Other kinds of bug generators are also discussed. We describe a representation both for novice natural language preprogramming knowledge and novice fragmentary programming knowledge. Using these representations and the bug generators, we evaluate the model by analyzing four interviews with novice programmers.

This paper is based on Jeffrey Bonar's doctoral dissertation.

Authors' present addresses: Jeffrey Bonar, Intelligent Tutoring Systems Group, Learning Research and Development Center, University of Pittsburgh, Pittsburgh, PA 15260; Elliot Soloway, Cognition and Programming Project, Computer Science Department, Yale University, New Haven, CT 06520.

CONTENTS

1. INTRODUCTION

There is a growing literature about bugs[1] in novice programming (Anderson, Farrell, & Sauers, 1984; Johnson, Draper, & Soloway, 1982; Kahney & Eisenstadt, 1982; Shneiderman, 1976; Soloway & Ehrlich, 1984). In this article, we propose a process model to explain bugs produced in early phases of an introductory course. We are concerned with the difficulties a novice has in using basic programming constructs in short programs. In particular, we focus on the knowledge that novice programmers bring to these early programming problems. Our key idea is that many novice programming bugs can be explained as inappropriate use of the knowledge used in writing step-by-step procedural specifications in natural language.[2]

[1]There are two usages for the term *bug*. We are using it to refer to an error in a person's behavior, particularly an error in a computer program he or she has written. Often, though, bug is used to refer to perturbations in a person's mental procedure for some task. In this usage, systematic errors in behavior are explained by the bugs in mental procedures. This second meaning is used by Brown and VanLehn (1980) and Resnick (1982) when discussing children's problems with multicolumn subtraction.

[2]Throughout, the term *natural language* will be used to refer to the language in which step-by-step procedures are written. *English,* the other obvious choice, was not used because it unnecessarily implied that the novice programming phenomena discussed here were limited to English.

Our model is motivated by patterns of behavior we repeatedly observed in video-taped interviews of novice programmers solving programming problems (see Bonar, 1985, for a complete discussion). In particular, we characterize what happens when a novice produces a bug:

> While solving a programming problem (writing a program), novices will encounter some aspect of the problem they don't understand (an impasse).
>
> In order to move beyond the impasse, novices cast about for a way to resolve the aspect of the problem they don't understand (a patch). Frequently, that resolution involves an appeal to their knowledge of natural language step-by-step procedures that would be applicable in a similar situation.
>
> In implementing the patch, a bug is introduced.

Consider an example from one of our interviews. Subject 13 is working on the averaging problem (Figure 1). This problem requires the student to read in a series of numbers, watching for an ending value of 99999 and producing the average of the numbers read in before the 99999. Producing the average requires the student to accumulate both a sum and count of the numbers read. In the excerpt of Figure 2, he has just specified the test for a **repeat until** loop and is considering how to implement the step after the loop. (Note that the most straightforward solution to this problem would use a **while** loop rather than the **repeat until** loop. We do not focus on that issue here, but see Soloway, Bonar, & Ehrlich, 1983.) At this point (beginning of Segment 143), we see the subject stumbling and apparently searching around. We call this commonly observed behavior an impasse. At the end of Segment 143, he proposes **then** as a connective between the loop test and the average computation after the loop. Note that he is unsure whether this is a correct approach (Segment 144). We describe this proposed **then** as a patch to the impasse. He tests this usage by sounding the phrase to himself (Segment 146), reasoning from Pascal's **if then** statement. Finally, in Segment 148, he uses phrasing from natural language to justify his usage of the buggy **repeat . . . until then . . . .** At this point, he has written the **then** and committed the bug.

Our model draws on the repair theory work of Brown and VanLehn (1980). They studied impasses in children's subtraction algorithms and developed a detailed theory and process model for how bugs arise and are patched (in their work, they refer to repairs instead of patches). In the more complex domain of programming, we have developed a model with two important components:

1. We characterize the knowledge that allows a novice to form a bridge between programming language syntax and semantics and higher level design

Figure 1. **The averaging problem.**

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

concerns. This is information about how the language constructs are used to accomplish standard programming tasks. We represent this information as schemalike structures called *programming plans*. We discuss the programming plans for both the introductory programming language Pascal and natural language step-by-step procedures.

2. We characterize impasses arising from missing or misapplied programming plans needed in the course of developing a program. We propose that many bugs arise out of novice strategies for patching an impasse and continuing a problem solution. We call these strategies *bug generators*. We focus on the bug generators that patch an impasse by using the knowledge of how the problem would be solved with natural language step-by-step procedures. This knowledge is used to to supply missing programming language knowledge.

The paper is organized as follows. In Section 2, we discuss programming plans as a representation to describe both the knowledge used to write step-by-step natural language procedures and the knowledge used to write Pascal programs. We also describe the relationship between natural language procedure knowledge and Pascal programming knowledge. Section 3 discusses the process of novices resolving impasses with patches characterized by our bug generators. In Section 4, we present a preliminary evaluation of the model.

2. REPRESENTING NOVICE PROGRAMMING KNOWLEDGE

The content and structure of novice knowledge has been studied in several different domains including physics (Chi, Feltovich, & Glaser, 1981; DiSessa, 1982), geometry (Anderson, Greeno, Kline, & Neves, 1981), algebra (Lewis, 1981; Matz, 1982), and programming (Ehrlich & Soloway, 1983; Rich, 1981; Soloway, Ehrlich, Bonar, & Greenspan, 1982). A key generalization is that novices use schemalike structures we call *plans*. We propose two kinds of plan knowledge and links between them:

Novice knowledge of step-by-step natural language procedures: We refer to the set of step-by-step natural language procedure plans that a novice

Figure 2. **Excerpt of transcript for Subject 13. Up to this point, the subject has written a loop body for the ending value averaging problem. Here he is working on specifying the test for the loop and the code following the loop. Square brackets enclose comments we have added.**

Subject 13: [reading from problem] After seeing **99999**, it should print out the correct average, so until, ahhh, until I equals **99999** [he writes **until I = 99999**].

Interviewer: Okay.

Subject 13: Ahhh, and then [he pauses, reading what he's written], "**repeat, until,**" and then I would have, then [writes **then**]. Can you have a **then** in the **repeat until**? No, I don't recall that.

Interviewer: Okay.

Subject 13: [mumbles] if then, while then, repeat until then [louder now], well, it makes sense. I'm not sure if that's right, I don't recall using it before.

Interviewer: How does it make sense?

Subject 13: It makes sense because you are repeating until that [points to **until** line] and *then* [his emphasis] what do you do? Well, *then* take the average. ... [after the **then** he writes **Average := Sum/N**]

---

brings to a programming course as *SSK* (step-by-step natural language programming knowledge).

Novice knowledge of the programming language under study: We refer to these plans as *PK* (Pascal programming knowledge).

Functional and surface links between the SSK and PK: These links allow a novice to traverse between the two different sets of knowledge structures.

In what follows, we describe each of these components in turn.

2.1. SSK: Preprogramming Knowledge

Novices bring a knowledge of standard tasks in step-by-step natural language procedures to their introductory programming course. Examples of such tasks include looping, making choices, and specifying sequences of actions. We have studied this preprogramming knowledge with problems like the factory gate problem shown in Figure 3. The problem asks the subject to write a

Figure 3. **The factory gate problem.**

Please write a set of explicit instructions to help a junior clerk collect payroll information for a factory. At the end of the next payday, the clerk will be sitting in front of the factory gates and has permission to look at employee pay checks. The clerk is to produce the average salary for the workers who come out of the door. This average should include only those workers who come out before the first supervisor comes out, and should not include the supervisor's salary.

step-by-step natural language procedure for a junior clerk to collect payroll information from workers coming out of a factory gate. The clerk needs to report on the average salary for all the workers who leave the gate before the first supervisor leaves the gate. This problem was designed to parallel the averaging problem.

Figure 4 shows a sample solution to the factory gate problem. This solution is from one of our subjects before he had taken a programming course, and it illustrates several features of step-by-step natural language procedures. Step 5, for example, specifies counting with the phrase "add number of"; Step 6 specifies a total with the phrase "add all." The overall structure of the loop (Steps 1 to 4) is specified by illustrating how to do the task for the first two workers followed by the phrase "and so on" (Step 3). Finally, step 4 specifies the stopping condition for the loop. This condition is phrased as a continuously active test, always watching the action of the loop for the exit condition to become true. (This is sometimes referred to as a *demon* control structure.) In general, the novice seems to be using plans for standard tasks such as adding up numbers, stopping the loop, and so forth.

When coming into a programming course, novices will have a fairly complete SSK. This is to be expected because SSK is regularly used by most people to specify simple step-by-step procedures (Bonar, 1985, discusses SSK plans in detail; see Miller, 1981, for an earlier discussion of some standard features in step-by-step natural language procedures).

2.2. PK: Fragments of Programming Knowledge

PK represents the knowledge that allows novices to write some parts of a program correctly. Besides the obvious information about syntax and semantics, PK plans also capture the goals and tactics critical to implementing some task as a program. Our plans contain several different pieces of information. Consider, for example, the counter variable plan shown in Figure 5 and its components:

Plans are linked together in a general/specific hierarchy. The counter variable plan, for example, is a specialization of the running total variable plan.

Figure 4. **Typical answer for the factory gate problem. This procedure was written by Subject 11 before studying Pascal.**

1. Identify worker, check name on list, check wages
2. Write it down
3. Wait for next worker, identify next, check name, and so on
4. When super comes out, stop
5. Add numbers of workers you've written down
6. Add all the wages
7. Divide the wages by the number of workers

Plans contain information about the key roles and aspects of the plan. In the counter variable plan, we represent this information as slots describing the plan, its initialization, its update, how it is used, and the type of the variable.

Plans contain knowledge needed to actually implement a plan in a specific programming language. With the counter variable plan, there is specific information about how a counter is implemented in Pascal.

Although a novice's SSK is relatively complete, his or her PK is fragmentary. This is to be expected; the novice is just learning about PK. Not only does a novice have fewer PK plans than would an expert, but the connections between the PK plans are not as rich.

2.3. Functional and Surface Links Between SSK and PK

Although we have discussed SSK and PK separately, they have many similarities. There are two in particular:

1. Functional similarities exist because both SSK and PK are concerned with repeated actions, choice between conditions, counting, and so on.

2. Surface similarities exist because the programming language Pascal (like most others) shares many words with natural language. There are many common lexical entities in the two plan sets, irrespective of functional similarity of the plans connected.

We capture these similarities as functional and surface links between the SSK and PK plan sets. Although we distinguish between these two kinds of links, novices often do not. For example, the bug in the protocol of Figure 2 shows a confusion between surface and functional links. We see that the subject is confused about the relationship between the natural language *then* (indicating a following step) and the **then** of Pascal's **if-then-else** construct. We say that he is

Figure 5. **Counter variable plan. The double right arrow (a) indicates a link to another (more general) plan. The boxes (☐) indicate slots of the plan.**

Counter Variable Plan

SPECIALIZATION OF ⇒ Running Total Variable Plan

- ☐ *Description:* Counts the occurrences of some specific action.
- ☐ *Initialization:* Set to zero.
- ☐ *Update:* Adds one to the current value.
- ☐ *How used?* Always found within a piece of code that is executed repeatedly.
- ☐ *Type:* Integer values.

Pascal Implementation of Counter Variable Plan

- ☐ *Initialization:* CV : = 0
- ☐ *Update:* CV : = CV + 1
- ☐ *How used?* Usually found within a **while** or **repeat** loop. Counts are created by successive increments as the events to be counted occur.
- ☐ *Type:* **integer**

---

using the surface (lexical) link between *then* and **then**. Based on this surface link, he is assuming a functional link that allows him to use **then** with a natural language meaning.

Another example from the protocols illustrates confusion between surface and functional links associated with the word *while*. In natural language, *while* is typically used as a continuously active test, as in: "*while* the highway stays along the coast, keep following it north." This kind of control structure is unusual in a programming language. More typical is a construct in which the loop condition gets tested once per loop iteration (e.g., the **while** loop in Pascal). The surface link between the two kinds of "while" allows a novice to infer similar semantics. One of our subjects even inferred a semantics for a continuously active test in the Pascal **while** loop: "every time [the variable tested in the **while** condition] is assigned a new value, the machine needs to check that value. . . ."

3. THE PROCESS OF GENERATING A BUG

Because novice programming knowledge is fragmentary, by definition there must be gaps in that knowledge. Thus, in writing a program, a novice will encounter such a gap and be at what we have called an impasse. In order to bridge these gaps, the novice uses patches. By their very nature, these patches are likely to be incorrect. For this reason, we call the processes that create these

patches bug generators. We argue that the bug generators draw on inappropriate or incorrect knowledge, such as from SSK.

Consider a detailed example of some bug generators. Subject 13 wrote **Sum : = 0 + I** as part of a running total update inside a loop body for the averaging problem (see his code in Figure 6). He has identified **I** as the variable to receive new values, entered by the user. He has also identified **Sum** as the variable to hold the running total. Although there are a number of problems with this code, we focus on a single bug. (In the Appendix we show a detailed excerpt from Subject 13's protocol. That excerpt shows Subject 13 actually writing this code and attempts to analyze all the bugs that appear there.) In the protocol, Subject 13 points to this line and says "it reads the sum." The only **Read** statement Subject 13 has used has no arguments and is above the loop along with a **Readln**. What, then, does the subject mean by "reading the sum"? Our bug analysis describes several relevant bug generators and is shown in Figure 7.

The impasse in this case is that Subject 13 did not know how to implement a read operation for an input new value variable in Pascal. A correct Pascal implementation uses an explicit **Read(I)** in each iteration of the loop. Three plausible explanations, each based on a different bug generator and one based on a slip interpretation, are shown for the bug. (The reason for multiple explanations is discussed later.)

The first bug generator plausibly explaining this bug is the *programming language used as it if were natural language (PL used as NL)* bug generator. This bug generator operates on a specific plan or plans, using surface links between SSK and PK versions of the plan. Even though they are surface links, the novice treats them as functional links, assuming that the natural language semantics can be used for programming language constructs. The programming language construct is given similar semantics to the parallel natural language construct. In the example with Subject 13, the PL used as NL bug generator operates on the input new value variable plan. Getting a new value for the input new value variable has been implemented implicitly, which we claim is typical of the natural language implementation.

This claim about natural language implementation derives from our studies of SSK using problems like the factory gate problem of Figure 3. In those studies, we found that getting data was often implemented implicitly. For example, in Line 1 of Subject 11's answer to the factory gate problem (Figure 4), he says: "1. Identify worker, check name on list, check wages." Here, we claim, getting the input value (the worker) has been done implicitly. Note that this operation is not always implicit. In Line 3 of the same factory gate procedure, for example, the subject says: "3. Wait for next worker, identify next, check name, and so on." In this case, we say that getting the input value has been done explicitly with the phrase: "Wait for next worker . . . ."

The second explanation for the bug in Figure 7 uses the *programming language interpreted as natural language (PL interpreted as NL)* bug generator operating on the

Figure 6. **Subject 13's first attempt at the loop for the averaging problem. Line numbers are referred to in the protocol shown in the Appendix. N is the counter variable, Sum is the running total variable, and I is the input new value variable.**

```
N := 0;                    (1)
Sum := 0;                  (2)
repeat                     (3)
    Sum := 0 + I           (4)
    N := 1                 (5)
    Sum := I + I           (6)
    N := 2                 (7)
until                      (8)
```

Read statement written above the loop. In this case, the subject is seen to be interpreting the **Read** as if it were a declaration statement. (There is support for this explanation in that the subject discussed the **Read** statement while discussing other declarations.) The patch was to interpret the **Read** as if it were natural language. We claim that in natural language one can declare that reading will be done and have that reading be implicit in the rest of the step-by-step procedure.

This claim about natural language again derives from our studies of SSK. This notion of an input declaration comes from a natural language implementation strategy where the subject uses an early step in the procedure to describe how new data values are retrieved. In later steps, new input values are referred to implicitly. For example, one of our subjects working on the factory gate problem used the lines: "Each worker will be coming out of the gate in turn. Write down the value of each paycheck . . . ."

Note that the distinction between the PL used as NL bug generator and the PL interpreted as NL bug generator is fine. In the PL interpreted as NL bug generator, the novice uses a programming language construct to implement a natural language plan: a **Read** used as a declaration that reading will occur into a certain variable. In the PL used as NL bug generator, on the other hand, the novice uses programming language constructs as if they had their natural language meanings or omits programming language constructs that would not be needed in natural language: the implicit **Read(I)** every time a data value is needed.

The third explanation for the bug in Figure 7 uses the *one variable assumed to have multiple roles (multirole variable)* bug generator. In this interpretation, the subject patches an impasse about getting new values in the loop by collapsing the purpose for two different variables. In particular, he has collapsed the running total to be accumulated with the new value to be read from a user. In this interpretation, he thinks that the running total happens automatically when a new value is read from the user. The subject understands the statement **Sum**

Figure 7. **Bug analysis showing the operation of bug generators. The hands pointing right (☞) indicate each different plausible bug generator explanation for the bug.**

BUG: "It reads the sum": Sum : = 0 + I

The subject should be saying something like "it reads in a value which is added into the sum."

☞ ***PL used as NL* on input new value variable**

The read operation is done implicitly whenever a value is needed.

☞ ***PL interpreted as NL* on Pascal—Read/ReadIn**

Here the subject is treating Read; ReadIn pair as if it was declaring that reading is to be done, and the results of the read will be used with sum.

☞ ***Multirole variable* on running total variable, input new value variable**

The subject is using the variable sum as if it will automatically get a new value added in whenever that new value is read.

☞ ***Slip***

Just slipped and forgot to say ". . . a value which is added. . . ."

: = 0 + I to mean that the variable Sum gets the value of I added in every time a new value of I gets Read. This bug generator derives from the flexibility of a natural language noun phrase. A noun phrase can have several different aspects, normally disambiguated by context. In the multirole variable interpretation of our example, the noun phrase "the sum" has the two aspects accumulated so far and made up of values read. Rosnick (1982) has found a similar collapsing of variable roles in algebra students solving word problems.

The fourth explanation is a slip bug generator explanation. It says that the subject simply spoke sloppily and clearly understands that a Read(I) statement must appear elsewhere.

From the example in Figure 7, we see that it is possible to produce plausible explanations for a bug based on patches from two or more bug generators. If several different patches produce the same result, there is no systematic way to choose between the different possible bug generators. In fact, it is reasonable that novices can construct a patch using several bug generators that suggest similar approaches. Given the current methodology, however, we have no way to relate the novice programmer's behavior to possible interactions between plausible bug generator explanations.

Notice that we describe each bug generator in the example in terms of the plans active at the time of the impasse. We view bug generators as procedures that use one or more active plans to create a patch. We have categorized the bug generators based on how they use the active plans. We describe each of these three categories, with examples of each.

SSK Confounds PK. With these bug generators, the novice uses an SSK version of an active plan to confound the PK version of that plan.

PL used as NL (programming language used as natural language) bug generators use a programming language construct because it has the same words as a phrase used in the natural language implementation of an active plan. For example, in Figure 7, Subject 13 may have used the programming construct **read** as implying an implicit read operation to be executed whenever needed. In natural language, data are often retrieved in this way.

PL interpreted as NL (programming language interpreted as natural language) bug generators interpret a programming language construct as if it were a phrase used in a natural language implementation of the plan on which it is operating. In Figure 2, for example, Subject 13 seems to have interpreted **then** from Pascal as the phrase *then* in natural language.

Multirole variable (multiple roles for a variable) bug generators use a single variable for multiple roles from current active plans. This bug is designed to capture that aspect of natural language where a single noun phrase can have several different referents depending on context. For example, in natural language we can talk about "the sum of the wages" and "the count of the wages." Similarly, one of our subjects seems to have used the same variable to stand for both a running total variable (summing input values) and the counter variable (counting those input values). (In the excerpt, **I** is the variable that holds the input values and **N** is the multirole counter variable and running total variable: "I want to get a statement that is going to be clear that we're going to add the numbers, each number entered, we'll have the tally of the, number of integers entered . . . ahhh, N equals, ahmmm integer." [Writes: **N := I**].)

NL construct (new programming language construct from natural language) bug generators invent a new programming construct based on a natural language implementation of the parameter plan. For example, one of our subjects wrote the following line: **New := next New;** to indicate that the next value is needed for the new value variable.

Generic name (variables named generically) bug generators use generic names for parts of the program or variables. These names are based on common programming language implementation strategies. Program elements are named on the basis of how we talk about them when describing the program. For example, our subjects would talk about the input new value variable as "the variable holding the integers read in" and give that variable the name **integer**.

Intra-PK. These bug generators use incorrect or incomplete PK versions of the active plan to patch an impasse. Patching occurs by using knowledge within PK.

Trace (statements ordered in execution order) bug generators order the program as if it is an execution trace. For example, one of our subjects produced the following code for the body of the averaging problem (**Sum** is the running total variable, **I** is the new value variable, and **N** is the counter variable):

```
Sum := 0 + I
N := 1
Sum := I + I
N := 2
```

Overgeneralize (programminng language overgeneralization) bug generators allow a subject to overgeneralize from one Pascal implementation plan to another. For example, subjects often will initialize all variables whether this is needed or not.

In tactically similar (tactical similarity) bug generators, the subject fails to distinguish between plans that do similar things but are implemented differently. For example, subjects will use the assignment operator to give a value to Pascal constants or in association with a **read** statement.

Other Confounds PK. With these bug generators, the novice uses knowledge from other (i.e., not SSK or PK) domains to confound the PK version of the active plan.

Other domain (knowledge from other domains) bug generators use an understanding from a domain such as mathematics. For example, one of our subjects assumed that the variable I will always increment implicitly, much like the i used in mathematical notation for a series.

OS confound (operating system confound) bug generators confound a command or operation from the operating system with a programming language construct. For example, one of our subjects used an editor command within his program.

Our bug descriptions also include a *slip* as a plausible explanation. A slip refers to a random error produced while the novice is distracted, a speech slip, or a typographical error.

4. EVALUATING THE MODEL

This section presents a preliminary evaluation of our model. In particular, we present evidence that a significant portion of novice bugs can be explained by the SSK confounds PK bug generators. Our data are taken from a detailed analysis of four protocols selected from a body of interviews originally conducted to explore novice programming cognition. In the selected protocols, we present the same introductory programming problem to four different novice programmers, all in the fourth to sixth week of an introductory programming course.

4.1. Methodology

In our interviews, we present introductory programming students with a Pascal programming problem. Subjects are instructed to think aloud as they develop a solution to this problem. Protocols of these sessions were then ana-

lyzed with standard techniques used to study novice understanding of physics (Chi et al., 1981), algebra (Clement, 1982), children's arithmetic (Resnick, 1982), and other domains. We have also drawn on methodologies developed by Newell and Simon (1972) and discussions of methodological issues arising from the use of verbal reports as data (Ericsson & Simon, 1980; Ginsberg, Kossan, Schwartz, & Swanson, 1981).

Our analysis of the protocols has two steps: the plan analysis and the bug analysis. In the plan analysis, we relate the relatively abstract plan descriptions to the actual utterances and programs produced by the novice. In addition, because our claim is that novices use both SSK and PK while programming, we need to know whether the novice has used an SSK or PK implementation of each plan. In order to make the distinctions between SSK and PK versions of each plan, we have developed a series of criteria for each. The criteria specify those subject utterances we use to justify a claim that a certain plan is in use.

The criteria for SSK plans are based on our analysis of a study in which 34 subjects were given problems like the factory gate problem (Figure 3). Each of these problems required the use of specific plans. A subject's response required some implementation for each of these plans. Each plan implementation found (including the case of not explicitly using the plan) is represented on the list of criteria for that plan. These lists of criteria represent the strategies we know to be used in implementing each plan. The criteria for PK plans are based on our own introspection about the implementation strategies for PK plans.

Consider the example of plan analysis and use of criteria in Figure 8. In that plan analysis, we claim that Subject 6 has used both an SSK and PK version of the results variable plan. In Segment 173, she is discussing her implementation of the results variable. In the first part of the segment, she describes the implementation without any reference to how the count and sum are accumulated. This satisfies a criterion for an SSK version of the plan. Later in the segment, she goes into detail about the accumulation of the count and sum, satisfying a criterion for a PK version of the plan. We call each such piece of evidence (i.e., satisfaction of a criterion) an *evidence item.*

An example in Figure 9 illustrates how subtle the plan analysis can get. In the plan analysis, we claim that Subject 11 has used both an SSK and PK version of the counter variable plan in a single sentence (Segment 112). Evidence for the SSK version is the phrase "counting the numbers of integers that come through." Evidence for the PK version is the phrase "implementing by ones." These evidence items suggest that the novice is using both the SSK and PK versions of the plan within the same sentence. The difference is based on how the counting operation is viewed. "Counting the number of integers that come through" indicates that the subject intends an operation that counts the elements of a set that has been completely acquired before the count starts. On the other hand, "incrementing by ones" both uses programming jargon ("incrementing") and indicates an operation that counts each element, one at a time.

Figure 8. **Protocol segment with claimed use of both an SSK and PK version of the results variable plan.**

Subject 6: . . . what I'll want to be doing at the end is dividing the sum by the total of numbers to find an average, so I'll always have to keep track of both. So, after each number is read in, I will have, I will be keeping track at that same time as the calculation is made of the count and of the sum, so at any one point where the sentinel's read in, I will have both figures available to read in an average.

PLAN: Result variable

Evidence: Natural language

". . . dividing the sum by the total of numbers to find an average, . . ."

Evidence: Pascal

"after each number is read in, . . . I will be keeping track . . . of the count and of the sum . . ."

Counts of evidence items are used in our quantitative summaries (presented later) to represent overall plan activity. Note that a plan instance with several evidence items actually counts once for each evidence item. We would count two evidence items in our example of Figure 9, one for an SSK version of the plan and one for a PK version.

In the bug analysis, we show how errors made by a novice can be plausibly explained as bug generators operating on currently active plans. We analyze each piece of buggy behavior in a separate bug annotation. In Figure 7, we showed an example of such an annotation. After briefly describing the bug, the annotation describes the bug generators and plans that plausibly explain the bugs.

The bug generator set used in our bug analysis was developed after an examination of the first protocol presented here. That bug generator set was then used to analyze the other three protocols. In this way, we provide a minimal test of the explanatory power of a chosen set.

4.2. Overview of Protocols Analyzed

Before presenting a detailed analysis of the subjects' performance, we present an overview of each subject's work on the averaging problem. These programs and descriptions provide a context for the data that follow.

Interview 1. Subject 13's final program is shown in Figure 10. There are a number of things wrong with this program, but most critical is his peculiar loop body. Notice that within the loop body, **Sum** (the running total variable) is first set to **0** (Line 1) and then to **I + Next I** (**I** is the input new value variable) (Line 2), while **N** (the counter variable) is set to **1** (Line 3) and then **2** (Line 4).

Figure 9. **Protocol segment with claimed use of both an SSK and PK version of the counter variable plan.**

Subject 11: . . . I want it simply incrementing by ones, it's counting the number of integers that come through, . . .

PLAN: Counter
Evidence: Natural language
"counting the number of integers . . ."
Evidence: Pascal
"incrementing by ones"

The subject seems to be implementing the loop with the following natural language strategy: Show an example of the first few steps and assume that the other iterations will happen correctly (he actually shows the first two steps). In the Appendix, we show an analyzed excerpt from Protocol 1.

Interview 2. Subject 6's final program is shown in Figure 11. It is almost correct. His problem is that there is no **READ** inside the loop. In the protocol, he convinces himself that he needs a **READ** above the loop (Line 1) to make the **WHILE** test make sense (Line 2), but never thinks to put a second **READ** inside the loop also. At one point, he uneasily states that each test of **NEWNUM** in the **WHILE** statement (Line 2) will know to read a new value for that iteration.

Interview 3. Subject 11's final program is shown in Figure 12. Her program is almost completely correct. Like Subject 6, her bugs are related to reading new values inside the loop. She recognized that some sort of **Read** was required inside the loop and even realized that it had to come after the **Count** increment (Line 1) and **Total** update (Line 2). The **Read (I)** below the loop (Line 3) was originally, and correctly, put at the bottom of the loop body. In the protocol, she argues that leaving it there would cause the program to read a new value before the previous value read is processed. She uneasily moved **Read (I)** out to its current position outside the loop (Line 3) and settled on **ReadIn** (Line 4) at the bottom of the loop to express, as she put it, "the right amount of reading."

Interview 4. Subject 12's final program is shown in Figure 13. This program is almost correct. Its bug involves the first value **READ** (before the start of the loop) (Line 1). She reads in a starting value, but then sets **NUM** (the input new value variable) to **0** (Line 2), losing that starting value. From the protocol, it seems that she does this because she wanted to initialize all variables used inside the loop to **0**.

Figure 14 contains overview statistics for the four protocols analyzed. Interviews lasted between 23 and 45 min. It is not clear why Subjects 2 and 4 took nearly twice as long as did Subjects 1 and 3. Time spent on the problem does not correlate with plan usage or with number of bugs.

Figure 10. Final program for Subject 13 working on the averaging problem. Numbers on the right mark lines discussed in the text.

```
Program Average (Input/, Output);
Var
   N, Sum : Integers
   Average : Real;
Const : Sentinel
Begin (* Average of the integers entered *)
   Writeln ('Enter series of integers to be averaged');
   Writeln (Integers will be Averaged when you
                                    enter the Integer 99999);
   Read;
   Readln;
   Sentinel := 99999;
   N := 0;
   Sum := 0;
Repeat
   Read;
   Readln;
   Sum := 0                                                    (1)
   N := 1                                                      (3)
   Sum := I + Next I                                           (2)
   N := 2                                                      (4)
   until I = 99999
   then Average = Sum/N
Writeln ('Average':= 0);
END.
```

Figure 11. Final program for Subject 6 working on the averaging problem. Numbers on the right mark lines discussed in the text.

```
PROGRAM AVERAGE (INPUT/, OUTPUT);
CONST
   SENT = 99999;
VAR
   NEWNUM, COUNT, SUM, AVE : INTEGER;
BEGIN
   COUNT := 0;
   SUM := 0;
   READLN;
   READ (NEWNUM);                                              (1)
   WHILE NEWNUM <> SENT                                        (2)
     DO BEGIN
        SUM := NEWNUM + SUM;
        COUNT := COUNT + 1;
        END;
   IF COUNT <> 0
      THEN AVE := SUM DIV COUNT
   WRITELN ('AVERAGE = '), AVE:8:2;
END.
```

Figure 12. Final program for Subject 11 working on the averaging problem. Numbers on the right mark lines discussed in the text.

```
Const
   Sentinel = 99999;
Var
   I, Count, total, AVG : Integers;
Begin
   Count := 0;
   Total := 0;
   Writeln ('Enter integer');
   Readln;
   Read (integer);
   While I <> 99999 Do
      Begin
         Count := Count + 1;                    (1)
         Total := Total + I                     (2)
         Readln                                 (4)
      End
   Read (I)                                     (3)
   Avg := Total Div Count
   Writeln ('Avg is ', Avg:0)
```

Figure 13. Final program for Subject 12 working on the averaging problem. Numbers on the right mark lines discussed in the text.

```
PROGRAM SUMUP (INPUT/,OUTPUT);
CONST
   SENTINEL = 99999
VAR
   NUM1, SUM, AVERAGE, COUNT : INTEGER;
BEGIN
   WRITELN ('READ IN AN INTEGER AND CONTINUE UNTIL');
           ('FINISHED THEN ENTER 99999 . . . ');
   READLN;
   READ (NUM);                                  (1)
   COUNT := 0;
   SUM := 0;
   NUM := 0;                                    (2)
   WHILE NUM <> SENTINEL DO
       BEGIN
       SUM := NUM + SUM;
       COUNT := COUNT + 1;
       READLN;
       READ (NUM)
       END;
AVERAGE := SUM DIV COUNT;
WRITELN ('AVERAGE IS, ' AVERAGE : 0);
END.
```

Figure 14. **Summary statistics for the four protocols analyzed. "Bugs" refers to the number of bugs found in the protocol.**

	Protocol Number			
	1	2	3	4
Subject number	13	6	11	12
Duration (minutes)	32	45	23	45
Plan evidence items	159	177	104	107
SSK Plans	43	50	16	16
PK Plans	116	127	88	91
Plan evidence items/minute	5.0	3.9	4.5	2.4
Bugs	32	19	11	10

5. RESULTS OF THE ANALYSIS

We present our analysis based on three expectations derived from our model of novice programming bugs.

5.1. Expectation 1: Regular and Extensive Use of Plans

We have claimed that novices use both SSK and PK represented as plans. If this is the case, plan usage should be pervasive: The first expectation is that novices show a regular and pervasive use of plans. Novices should make use of both SSK and PK plans.

Figure 14 contains the statistics relevant to this expectation. There were 104 plan evidence items in the protocol with the fewest plan evidence items (Protocol 3). This protocol averaged 4.5 plan evidence items per minute. The protocol with the most evidence items had 177, averaging 3.9 plan evidence items per minute. Although there is no base line, it seem fair to say that our subjects did use plans pervasively.

In Figure 15, we show the plan evidence item counts broken down by type of plan (with SSK and PK lumped together). From this fine-grained breakdown, we see that all subjects used every type of plan. More intriguing, however, is that this breakdown permits an interesting speculation. Consider the following data about several different plans:

Figure 15. **Plan evidence items for the analyzed protocols.**

	Protocol Number			
	1	2	3	4
Sentinel	9 (5%)	8 (4%)	4 (4%)	11 (10%)
Loop	20 (13%)	13 (7%)	9 (9%)	1 (1%)
Counter	29 (18%)	21 (12%)	18 (17%)	12 (11%)
Arithmetic sum	21 (13%)	38 (21%)	13 (13%)	15 (14%)
Result	23 (14%)	18 (10%)	9 (9%)	12 (11%)
New value	11 (7%)	13 (7%)	16 (15%)	16 (15%)
Input	18 (11%)	21 (12%)	15 (14%)	8 (7%)
New value loop	9 (6%)	35 (19%)	14 (13%)	23 (21%)
Result output	10 (6%)	6 (3%)	0 (0%)	1 (1%)
Instructional output	7 (4%)	0 (0%)	3 (3%)	0 (0%)
Prompt output	2 (1%)	1 (1%)	3 (3%)	8 (7%)
Illegal filter	0 (0%)	3 (2%)	0 (0%)	0 (0%)
Total	159 (100%)	177 (100%)	104 (100%)	107 (100%)

New value controlled loop plan: In Protocol 1, where the subject's loop was very far from correct, the count is quite low, accounting for only 6% of the plan evidence items. Other subjects, whose loops were fairly close to correct, had higher counts, ranging between 13% and 21% of all plan evidence items.

Sentinel variable plan: All subjects had little trouble with this plan and all had relatively low plan evidence item counts, ranging between 4% and 10% of all plan evidence items.

Input new value variable plan and input plan: These plans gave all subjects a great deal of trouble. Plan evidence items counts (for the two together) range between 18% and 33% of all plan evidence items.

From these and similar observations, it appears that plan activity is highest on those plans that a novice almost knows (i.e., those plans where the novice has a fairly good idea of what is needed, but does not have all aspects correct).

Figure 16. **Bug generator statistics for the analyzed protocols. "Bugs explained" refers to the number of bugs that can be explained by one or more bug generators developed from the analysis of Protocol 1. "Total explanations" refers to the number of plausible, nonslip bug generator explanations. "Explanations/bug" refers to the average number of plausible, nonslip explanations per bug.**

	Protocol Number			
	1	2	3	4
Bugs found	32	19	11	10
Bugs explained	—	19	11	10
Total explanations	55	27	15	15
Explanations/bug	1.7	1.4	1.4	1.5

5.2. Expectation 2: Bug Generators Plausibly Explain the Bugs

The second expectation is that the bug generator set presented here can plausibly explain most errors found in the protocols. The data relevant to this expectation appear in the bugs explained line of Figure 16. As discussed earlier, the bug generator set was developed based on the analysis of Protocol 1. The data indicate that one or more bug generators from that set can plausibly explain each of the bugs found in Protocols 2, 3, and 4.

It is important that the bug generators actually do discriminate between the bugs. In our data, this would mean that there are relatively few plausible explanations per bug. Figure 16 shows the total number of plausible explanations in each protocol and the mean number of plausible explanations per bug. These data indicate that although multiple bug generators may be responsible for some bugs, on the average less than two bug generators can plausibly explain each bug.

The plausible bug generator explanations summarized in Figure 16 do not include cases where the bug was plausibly explained as a slip. These data are summarized in Figure 17. First, notice that all but two bugs had a plausible nonslip explanation. That is, almost every bug detected in the analysis can be plausibly explained as a bug generator operating on plans. Second, notice that slips can plausibly explain between 34% and 58% of the bugs (depending on the protocol). Most likely, a much smaller percentage are actually slips. Though there are many bugs that can plausibly be interpreted as slips, many of these slip interpretations become less likely when the bug is examined in the context of the whole protocol. That is, subjects often return to a bug and explicitly discuss it. It is unlikely that a slip would stand up to such repeat scrutiny.

Figure 17. **Slip versus nonslip bug generator statistics.**

	Protocol Number			
	1	2	3	4
Bugs plausibly explained by nonslip bug generators	32 (100%)	18 (95%)	10 (91%)	10 (100%)
Bugs plausibly explained by slip bug generators	11 (34%)	11 (58%)	5 (45%)	5 (50%)

5.3. Expectation 3: SSK/PK Bug Generators Are Critical

In our model, we propose that the intrusion of SSK plays an important role in novice programming bugs. The third expectation is that when novices encounter an impasse in a developing programming solution, they usually use SSK confounds PK bug generators to patch and continue. The analysis presented here contains two kinds of evidence for this expectation.

First, SSK confounds PK bug generators embody the process whereby novices use SSK knowledge to reason about their programs. As can be seen in the SSK/PK total line of Figure 18, SSK confounds PK bug generators plausibly explained 60%, 67%, 67%, and 47% of the bugs found in Protocols 1 to 4, respectively. That is, for Protocols 1 to 3, SSK confounds PK bug generators can explain 60% or more of the bugs. Protocol 4 had the lowest (47%) coverage by SSK confounds PK bug generators but also had the least buggy protocol.

Second, the subjects had between 10 and 32 bugs. Notice that in terms of bugs, there seem to be two groups in the protocols: Protocols 1 and 2 have more bugs than Protocols 3 and 4. We refer to these as the "buggy group" and the "less buggy group," respectively. Referring to Figure 19, we see that the buggy group had higher ratios of SSK to PK plan evidence items than the less buggy group. The ratios were .37 and .39 for the buggy group protocols and .18 for both protocols of the less buggy group. This is consistent with the notion that a more error prone and less advanced novice is doing more reasoning from SSK plans.

6. CONCLUDING REMARKS

In the preceding section, we presented an analysis of four protocols of novice programmers. From that analysis we provide data to support our model of novice programmer bugs. Specifically, we present evidence that:

> Plans, both SSK and PK, are pervasive in the work of novice programmers.

Figure 18. **The number of bugs plausibly explained by each nonslip bug generator, plus the percentage of plausible explanations by the specified bug generator out of all possible nonslip explanations. Each bug can have several explanations. Slip data are presented in Figure 17.**

	Protocol Number			
	1	2	3	4
SSK Confounds PK:				
PL used as NL	10 (18%)	6 (22%)	1 (7%)	1 (7%)
NL interprets PL	11 (20%)	10 (37%)	7 (47%)	3 (20%)
Multirole	8 (15%)	1 (4%)	1 (7%)	3 (20%)
NL construct	2 (4%)	0 (0%)	0 (0%)	0 (0%)
Generic name	2 (4%)	1 (4%)	1 (7%)	0 (0%)
Total	33 (60%)	18 (67%)	10 (67%)	7 (47%)
Intra-PK:				
Trace	10 (18%)	5 (19%)	1 (7%)	1 (7%)
Overgeneralize	4 (7%)	1 (4%)	2 (13%)	3 (20%)
Similar	4 (7%)	1 (4%)	2 (13%)	3 (20%)
Total	18 (32%)	7 (26%)	5 (33%)	7 (47%)
Other Confounds PK:				
Other domain	3 (5%)	2 (7%)	0 (0%)	1 (7%)
OS confound	1 (2%)	0 (0%)	0 (0%)	0 (0%)
Total	4 (7%)	2 (7%)	0 (0%)	1 (7%)
Total	55 (100%)	27 (100%)	15 (100%)	15 (100%)

Our bug generator set, developed after studying only one protocol, can plausibly explain the novice bugs observed in the other protocols.

Most important, SSK plays an important role in the bugs of novice programmers. Bug generators that describe the confounds between SSK and PK can plausibly explain a substantial number of bugs in each of the protocols. In addition, the SSK to PK plan ratio was higher for those novices with buggier protocols.

Figure 19. **PK to SSK plan usage ratios support the importance of SSK in understanding buggy programs.**

	Protocol Number			
	1	2	3	4
Plan Evidence Items	159	177	104	107
SSK plans	43	50	16	16
PK plans	116	127	88	91
Ratio: PK to SSK	.37	.39	.18	.18

There are several next steps planned for this work. Currently, our bug analysis does not provide data to support our specific set of bug generators, nor does it provide the ability to recognize which of the plausible bug generators actually contribute to the bug. We would like to develop the theory that makes this more specific: Are there ways to detect exactly which bug generator(s) is (are) responsible? This will require a more formal definition of each bug generator along with a more constrained set of interviews. In these interviews, subjects will be put in specific situations where bugs are very likely. For example, subjects might be given a problem statement and a skeleton program missing a key section. Such interviews would be accompanied by probes to detect the use of certain bug generators and plans. We plan to focus particularly on the input new value variable plan's use within a loop, an area of difficulty for all our subjects. In general, correctly updating within the loop seems to be difficult for our subjects and amenable to more constrained study.

We are planning to explore the use of SSK plans in better understanding the intentions of novice programmers. We are beginning the development of a curriculum organized around common programming plans. Each programming plan would be introduced by examining the SSK version. The SSK plans allow us to understand and direct the students in their basic approach to the problem solution. With the SSK plan providing a connection to something the student already knows, we then introduce the PK version of the plan and actual code. By making a clear distinction between the SSK and PK versions, we should be able to avoid many of the errors discussed in this article.

We have presented a process model of bugs for novice programmers early in a programming course. In developing our model, we have focused on the knowledge used by a novice programmer. In particular, our model is based on the role of natural language step-by-step procedural knowledge in understanding these bugs. Although our model is just a start toward understanding the source of novice bugs, it indicates how critical novice preprogramming knowledge is likely to be in a more complete theory.

Acknowledgments. The authors wish to thank the referees for their valuable comments and careful reading.

Support. This work was supported by the National Science Foundation under NSF Grant MCS-8302382. Currently, the first author is supported by the Office of Naval Research under Contract Numbers N00014-83-6-0148 and N00014-83-K-0655. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors and do not necessarily reflect the views of the U.S. government.

REFERENCES

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science, 8,* 87-129.

Anderson, J. R., Greeno, J. G., Kline, P. J., & Neves, D. M. (1981). Acquisition of problem-solving skill. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 191-230). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Bonar, J. G. (1985). *Understanding the bugs of novice programmers.* Unpublished doctoral dissertation. University of Massachusetts, Amherst.

Brown, J. S., & VanLehn, K. (1980). Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4,* 379-426.

Chi, M. T., Feltovich, P., & Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. *Cognitive Science, 5,* 121-152.

Clement, J. (1982, January). Students' preconceptions in introductory mechanics. *American Journal of Physics, 50,* 66-71.

DiSessa, A. A. (1982). Unlearning Aristotelian physics. *Cognitive Science, 6,* 37-76.

Ehrlich, K., & Soloway, E. M. (1983). An empirical investigation of the tacit knowledge in programming. In J. Thomas & M. L. Schneider (Eds.), *Human factors in computer systems.* Norwood, NJ: Ablex.

Ericsson, K. A., & Simon, H. (1980). Verbal reports as data. *Psychological Review, 87,* 215-251.

Ginsberg, H. P., Kossan, N., Schwartz, R., & Swanson, D. (1981). Protocol methods in research on mathematical thinking. In H. P. Ginsberg (Ed.), *Development of mathematical thinking* (pp. 7-47). London: Academic.

Johnson, W. L., Draper, S., & Soloway, E. M. (1982). Classifying bugs is a tricky business. In *Proceedings of the Seventh Annual NASA/Goddard Workshop on Software Engineering.*

Kahney, H., & Eisenstadt, M. (1982). Programmers' mental models of their programming tasks: The interaction of real-world knowledge and programming knowledge. *Proceedings of the Fourth Annual Conference of the Cognitive Science Society,* pp. 143-145.

Lewis, C. (1981). Skill in algebra. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 85-110). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Matz, M. (1982). Towards a process model for high school algebra errors. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 25-50). London: Academic.

Miller, L. A. (1981). Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal, 20,* 184-215.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Resnick, L. B. (1982). Syntax and semantics in learning to subtract. In T. Carpenter, J. Moser, & T. Romberg (Eds.), *Addition and subtraction: A cognitive perspective* (pp. 136-155). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Rich, C. (1981). *Inspection methods in programming* (Tech. Rep. No. AI-TR-604). Cambridge: MIT, MIT Artificial Intelligence Laboratory.

Rosnick, P. (1982). *Student conceptions of semantically laden letters in algebra.* Amherst: University of Massachusetts, Cognitive Development Project.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences, 5,* 123-143.

Soloway, E. M., Bonar, J. G., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the Association for Computing Machinery, 26* (11), 853-860.

Soloway, E. M., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions of Software Engineering, SE-10,* 595-609.

Soloway, E. M., Ehrlich, K., Bonar, J. G., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), *Directions in human computer interaction* (pp. 27-54). Norwood, NJ: Ablex.

HCI Editorial Record. First manuscript received November 7, 1984. Revision received April 26, 1985. Accepted by John Seely Brown. Final manuscript received May 3, 1985. —*Editor*

APPENDIX: A SAMPLE OF AN ANALYZED PROTOCOL

This appendix contains a short segment from a fully analyzed protocol. This segment is taken from Protocol 1 in the set of four protocols analyzed in depth and discussed in Section 5 of the text.

The subject of this segment is working on the averaging problem. At the point we pick up the protocol, he is coding the loop. The following transcript is annotated with the plan analysis and the bug analysis. In the course of the protocol, the subject will write the code shown in Figure 6. The subject's basic problem is that he does not understand how variables are used to process values inside the loop. As we see in the protocol, the subject clearly understands that the loop body must accumulate a running total in the variable **Sum** and a count in the variable **N**. He also understands that the variable **I** will hold new values read from the user. Explanations for the peculiar code in the loop body are presented in the bug analysis annotations with the actual transcript segment.

The transcript uses the following notational conventions:

Square brackets (e.g., [ ]) enclose explanatory comments we have added to the text.

Programming font (e.g., **N := 1**) is used to indicate code that the subject actually wrote during the protocol.

Plan annotations are set off from the protocol text with horizontal lines and labeled "PLAN:". These annotations mark those places where we claim the subject has used a plan. The annotation describes our evidence for that claim. See Section 4.1 for a detailed discussion of the plan annotations.

Bug annotations are set off from the protocol text with horizontal lines and labeled "BUG:". These annotations mark those places where the subject has a bug. The annotation provides a general explanation of the bug and describes the bug generators that could plausibly explain that bug. Each plausible bug generator explanation is marked with a hand pointing right (☞). See Section 4.1 for a detailed discussion of the bug annotations.

Here is the protocol segment:

Subject 13: Ummm, ummm, well I think it's, ahh, it's [N := I, eventually changed to N := 1, eventually put on Line 5] not right I don't think, but I, I'm gonna leave it that way for the moment.

Interviewer: Okay, fine.

Subject 13: And then integer [the way the subject refers to the variable I], or rather, sum equals integer, ahm, equals zero plus integer [Writes: Sum := 0 + I, eventually put on Line 4], and the number equals the integer, ahhh.

PLAN: Arithmetic sum variable
Evidence: P3—Running total assignment to Sum in loop.

BUG: Arithmetic sum set to current new value variable
There will be no running sum operation because the arithmetic sum, Sum, always gets the value of the current new value variable, I.

☞ ***Trace* on arithmetic sum variable**
The subject is reasoning about the execution behavior of the arithmetic sum variable. He recognizes that on the loops first iteration, Sum will be given the value 0 + I and then writes that.

Interviewer: Why don't you tell me what you are thinking?

Subject 13: Well, I'm thinking that [points to N := 1, eventually changed to N := 1 and eventually put on Line 5] should go after [points to Sum := 0 + 1, eventually put on Line 4] because the sum is going to be zero plus the integer and then the number is going to be, ahh, number equals 1.

Interviewer: Ah huh.

Subject 13: [Crosses off N := 1] And then number equals one [Writes: N := 1 on Line 5].

PLAN: Counter variable
Evidence: N7—Counting to N done after operation.

BUG: Counter needs to go after the arithmetic sum
The increment of the counter variable, N, needs to be done after the update of the arithmetic sum variable, Sum.

☞ ***PL interpreted as NL* on counter variable**
In our study of natural language step-by-step procedures, we found that the counting operation is usually specified after the operation to be counted. Also, in one of our interviews, the subject suggested that if counting happened before the operation to be counted, there was a chance that the count would be 1 too high. Here the subject is applying that convention to Pascal.

Interviewer: Okay.

Subject 13: And then, and then sum equals integer plus integer [Writes: Sum := I + I on Line 6], and number equals 1. Ahhh.

PLAN: Arithmetic sum variable
Evidence: P3—Running total assignment to Sum in loop.

BUG: Arithmetic sum set to current new value variable
There will be no running sum operation because the arithmetic sum variable, Sum, always gets twice the value of the current new value variable I.

☞ ***Trace* on arithmetic sum variable**
The subject is reasoning about the execution behavior of the arithmetic sum variable. He recognizes that on the loops second iteration, Sum will be given the value I + I and then writes that.

☞ ***Multirole variable* on new value variable**
The subject is allowing the new value variable to take two roles: its "sum so far" and its next value. Note that later in the transcript, he will add *next* as a keyword in front of the second I.

Interviewer: What are you thinking now?

Subject 13: Number equals 2 [Writes: N := 2 on Line 7] and it would go on, it would repeat, that, if [the loop body] continues to repeat [sweeping motions]

this [points to the 2 on Line 7] will increase. I'm assuming for the moment that this is sufficient input.

PLAN: Counter variable
Evidence: P3—Increment counter variable N inside the loop

PLAN: Indefinite Loop
Evidence: N1—"Continues"
Evidence: N2—"*this will repeat until . . .*"

BUG: Says counter variable will increase, but it won't
Says that the counter variable, N, will increase based on the N := 1 and N := 2 inside the loop.

***PL interpreted as NL* on Pascal—repeat, indefinite loop**
The subject is expecting the loop to work like loops in natural language. There, it is common to specify a loop by giving one or two cases of the iteration and assuming that the person reading will know how to generalize. Notice below () he says that Sum := 0 + 1 (Line 4) is the "first format of that," referring to the action performed for each value of the new value variable.

Interviewer: Okay, "sufficient input"?

Subject 13: Input to [pause] so that the computer will know that, for each [pause] for each integer entered, you add 1, you add the integer to the sum [points to Sum := 0 + 1 on Line 4], and that this is the first format of that, zero plus integer, N equals 1, sum equals integer plus integer, number = 2 ["next" motion with hand], until [pause] [Writes: until].

PLAN: Indefinite loop
Evidence: N3—"for each . . ."

PLAN: Counter variable
Evidence: P3—Increment counter N inside the loop

PLAN: Arithmetic sum variable
Evidence: P3—Update arithmetic sum, Sum, inside the loop

This concludes the example protocol analysis.