SESSION XI MAN-MACHINE INTERACTIONS

Collecting and analyzing on-line protocols from novice programmers

JEFFREY BONAR

*Computer and Information Science Department, University of Massachusetts
Amherst, Massachusetts 01003*

KATE EHRLICH and ELLIOT SOLOWAY

Computer Science Department, Yale University, New Haven, Connecticut 06520

and

ERIC RUBIN

*Computer and Information Science Department, University of Massachusetts
Amherst, Massachusetts 01003*

Methodology for collecting and analyzing on-line protocols from novice programmers is described. On-line protocols are copies of all syntactically correct programs that students have written using an interactive computer system. Since the number of on-line protocols collected is quite large, we have developed a computer program, called the Bug Finder, which can automatically identify semantic and pragmatic bugs in subjects' programs. In this paper, we describe the theory upon which the Bug Finder is built and provide an example of the Bug Finder in operation.

The cognition and programming group at Yale University has been examining the programming skills and abilities of novices and experts with the intention of building MENO-II, a computer-based intelligent tutoring system for novices (see Soloway, Woolf, Rubin, & Barth, 1981). In order to build such a system, we have empirically studied the errors and misconceptions of novice programmers. One form of data that is quite valuable is "on-line protocols." Our on-line protocols are copies of every syntactically correct program¹ run by a student. In order to process the large number of programs gathered in this manner, we have designed and built a computer-based automatic analysis system, the Bug Finder, that can detect semantic and pragmatic bugs in subjects' programs. In this paper, the methodology used to collect and analyze these on-line protocols is described.

This work was supported by the Army Research Institute for the Behavioral and Social Sciences, under ARI Grant MDA903-80-C-0508, and by the National Science Foundation, under NSF Grant SED-81-12403. 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. Requests for reprints should be addressed to Jeff Bonar, Computer and Information Science Department, University of Massachusetts, Amherst, Massachusetts 01003.

In order to illustrate the kind of data we are dealing with and the range of the Bug Finder's capabilities, we have displayed in Figure 1 the actual output of the Bug Finder on one typical program. The problem, for which the program depicts an attempted solution, is to perform some standard numerical operations on a set of numbers (the exact problem is also given in Figure 1).

This problem is illustrative of the problems used in introductory programming classes.² The program depicted in Figure 1 has a number of semantic and pragmatic bugs. These bugs are identified in the output following the program; this output was actually produced by the Bug Finder. For instance, Bug 2 is an example of a semantic bug; here, the student did not change the value of the loop control variable, RAINFALL, in the loop and thus created an "infinite loop bug." Bug 4 is an example of a pragmatic bug. While the statement in Line 15 is syntactically and semantically correct, it does not "make sense" in the context of this problem. (The correct statement should be "TOTAL := TOTAL + RAINFALL.")

Note carefully that the Bug Finder can be used in two modes. First, it can be used in batch mode, in which a semester's worth of programs are input to it all at one time. Second, it can be used on-line as a tutor; in this mode, a student submits his program to the Bug Finder

Problem: Read in numbers, which represent the amount of daily rainfall, until the sentinel value 99999 is read. Print out: (1) the number of valid rainfall days (a negative value can be input and should not be counted as rainfall), (2) the average of all valid rainfalls, (3) the highest daily rainfall, and (4) the total number of rainy days (i.e., a day where rainfall was greater than 0).

```

1 PROGRAM RAIN (INPUT/OUTPUT);
2  CONST SENTINAL = 99999;
3  VAR COUNT : INTEGER;
4      RAINFALL, TOTAL, AVERAGE, HIGHEST : REAL;
5  BEGIN
6      COUNT := 0; TOTAL := 0;
7      WRITELN ('ENTER RAINFALL FOR ONE DAY');
8      READLN; READ (RAINFALL);
9      WHILE RAINFALL <> SENTINAL DO
10         BEGIN
11             COUNT := COUNT + 1;
12             TOTAL := TOTAL + RAINFALL;
13             IF RAINFALL > HIGHEST
14             THEN HIGHEST := RAINFALL;
15             WRITELN ('ENTER RAINFALL');
16         END;
17     IF RAINFALL = SENTINAL
18     THEN AVERAGE := TOTAL / COUNT;
19     WRITELN (COUNT, ' VALID RAINFALLS WERE ENTERED');
20     WRITELN ('THE AVERAGE WAS ', AVERAGE);
21     WRITELN ('THE HIGHEST WAS ', HIGHEST)
22 END.

```

1. **** UNINITIALIZED VARIABLE BUG ****
THE IF STATEMENT STARTING ON LINE 16 REFERS TO THE VALUE OF HIGHEST BUT AT THIS POINT HIGHEST HASN'T GOTTEN A VALUE YET.
2. **** INFINITE LOOP BUG ****
THE WHILE LOOP STARTING ON LINE 12 IS AN INFINITE LOOP. THE VARIABLE RAINFALL IS NOT MODIFIED IN THE BODY OF THE LOOP, SO IF THE CONDITION RAINFALL <> SENTINAL IS TRUE ON EXECUTING THE LOOP IT WILL NEVER BECOME FALSE, AND THE LOOP WILL EXECUTE FOREVER.
3. **** REDUNDANT TEST BUG ****
THE IF STATEMENT ON LINE 20 TESTS WHETHER RAINFALL = SENTINAL, BUT AT THIS POINT IN THE PROGRAM THIS CONDITION HAS TO BE TRUE, BECAUSE IT IS ONLY POSSIBLE TO GET OUT OF THE WHILE LOOP STARTING ON LINE 12 WHEN THE CONDITION RAINFALL <> SENTINAL IS FALSE.
4. **** ADD IN BUG ****
IT SEEMS THAT YOU ARE TRYING TO ADD THE RAINFALLS INTO TOTAL BY ADDING TOTAL TO ITSELF IN THE ASSIGNMENT STATEMENT ON LINE 16.
5. **** DIVIDE BY ZERO BUG ****
YOU HAVE NOT CORRECTLY CHECKED THAT COUNT IS NOT ZERO BEFORE DIVIDING BY COUNT ON LINE 21. YOUR PROGRAM WILL NOT HANDLE THE CASE WHEN NO VALID RAINFALLS ARE ENTERED.

Figure 1. Example use of the Bug Finder/Tutor.

and immediately receives its analysis. While we are using the Bug Finder in both modes, the data reported in this paper were generated when the Bug Finder was used as a tutor.

In the next section, we discuss the reasons why on-line protocol data are needed.

THE NEED FOR ON-LINE PROTOCOLS

Although written group tests have produced many interesting results, tests of this type are not always satisfactory. Written tests remove students from the normal environment for programming. Typically, programming students have a computer available on which to test and evolve their programs. Hence it is important to obtain data from subjects who are engaged in programming (e.g., from students who are attempting to write programs to solve their homework assignments).

In order to capture this sort of data, we built a special subsystem, CNS122, on a CDC CYBER 172. This subsystem permitted the normal programming functions (e.g., editing, compiling) to be carried out and, in addition, automatically sent a copy of all syntactically correct programs onto permanent storage (magnetic tape).

Besides reflecting a more normal setting for programming, these data give us a better glimpse into the programmer's intermediate thought processes. Whereas on a written test all we see is the "final output," on-line protocols capture the whole history of the program's development. These data allow us to answer the following sorts of questions: Catalogue of bugs—Are there a large number of programming "bugs"? Can we describe and categorize them? Initial bugs and persistent bugs—Are there bugs that occur early in the development of a program but are easily resolved by novices? In contrast,

are there bugs that persist and resist correction as novices continue to work on that program? Pattern of bugs over the semester—Do the bug types change as a novice gains experience? Are there some particularly insidious bugs that plague students throughout the semester? Stereotypic correction patterns—Do students attempt to resolve bugs in recognizably specific ways? What new bugs are typically introduced in these changes? Program styles—Even for small problems, are there a variety of program solutions? How do these compare with those of an expert?

THE KNOWLEDGE USED IN PROGRAMMING: A PRELIMINARY THEORY

The key to the Bug Finder's success is its use of high-level programming knowledge to abstract the critical factors in a program. We call this sort of information plan knowledge. Briefly, a plan is an abstraction that captures the essential components of a process. Moreover, it appears that expert programmers have and use an extensive network of just such plan knowledge about programming (see Adelson, 1981; Shneiderman, 1976). We have identified a number of common variable plans and looping plans found in programming. (For an extensive discussion of programming plans, see Soloway, Bonar, Woolf, Barth, Rubin, & Ehrlich, 1981; Soloway, Ehrlich, Bonar, & Greenspan, 1982.)

Consider the following 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. That is, it should not count the final 99999."

Figure 2 shows a program that solves this problem and illustrates a number of plans.

The variable COUNT (see Figure 3) is used to keep track of the number of numbers read in; since it serves the role of a counter, we call COUNT a counter variable. Similarly, SUM is used to accumulate a running total; hence we call SUM a running-total variable. The loop structure is an example of a new-value controlled running-total loop plan: (1) The objective of the loop is to accumulate a total (in SUM), and (2) the loop will

```
PROGRAM EXAMPLE-OF-PLANS;
  VAR COUNT, SUM, NUMBER : INTEGER; AVERAGE : REAL;
  BEGIN
    COUNT := 0;
    SUM := 0;
    READ (NUMBER);
    WHILE NUMBER <> 99999 DO
      BEGIN
        SUM := SUM + NUMBER;
        COUNT := COUNT + 1;
        READ (NUMBER)
      END;
    AVERAGE := SUM / COUNT;
    WRITELN (AVERAGE)
  END.
```

Figure 2. A program that illustrates common plans.

```
1 PROGRAM BUGGY-EXAMPLE (INPUT/,OUTPUT);
2   VAR
3     SUM, NEW : REAL;
4   BEGIN
5     WHILE NEW <> 99999
6     BEGIN
7       READ(NEW);
8       SUM := SUM + NEW
9     END;
10    WRITELN(SUM);
11  END.
```

Figure 3. A buggy program that is an attempt to solve the problem presented in the text. The variable SUM is not initialized to 0. Similarly, the variable NEW has no value the first time it is tested at the top of the WHILE loop. Finally, notice that this program will add in the final 99999.

terminate when the new-value variable (NUMBER in this case) attains a specified value (99999). Note that these plans are common to many programs. Typically, a programmer weaves together many such plans when writing a large program.

COLLECTING AND ANALYZING THE DATA: THE BUG FINDER

The CNS122 system appears to the student as a subsystem on a CDC CYBER 172 under the NOS operating system. In CNS122, we have redefined the actions of several standard system commands such as "RUN" (the command that actually runs a program under development). While sometimes this is done to simplify what a novice needs to know to use the system, it is primarily done so that copies of the syntactically correct novice programs can be saved without requiring any extra work from the students. (This procedure is carried out only with students who have given their prior consent.) These saved programs constitute the on-line protocols. In addition to what the students use, we have developed a package of utilities for backing up student programs to tape and overcoming inadequacies of the base operating system. CNS122 has been in use for several semesters.

The only effective way to analyze the amount of on-line protocol data we have collected (literally thousands of programs) is to have the computer itself do the brunt of the work. In order to find semantic and pragmatic bugs, the analysis program must "understand" the student's program. For example, while a student's program may actually execute and produce answers, the program may still not solve the given problem. General program understanding is an exceedingly difficult problem. In fact, there is a whole area of artificial intelligence devoted specifically to this research topic. Our approach to this problem is to constrain the types of programs the Bug Finder can accommodate and to constrain the range of bugs that can be found. In particular, the Bug Finder has been tailored to identify only those common bugs that have been previously

```

1 PROGRAM CORRECT-EXAMPLE (INPUT/,OUTPUT);
2 VAR
3   SUM, NEW : REAL;
4 BEGIN
5   SUM := 0;
6   READ(NEW);
7   WHILE NEW <> 99999
8     BEGIN
9       SUM := SUM + NEW;
10      READ(NEW)
11    END;
12   WRITELN(SUM);
13 END.

```

Figure 4. This program correctly solves the problem presented in Section 3.

isolated in empirical studies. (For detailed discussions of some of these bugs, see Soloway, Bonar, Woolf, Barth, Rubin, & Ehrlich, 1981; Soloway et al., 1982.)

We will describe the four stages in the bug finding process in the context of an example. In particular, we will analyze the program in Figure 4. This program is an attempt to solve the following problem. "Read in numbers, taking their sum, until the number 99999 is seen. Report the sum. Do not include the final 99999 in the sum." (A correct version of that program is also given in Figure 4.) Notice that there are several bugs in this program. The variable SUM is not initialized to 0. Similarly, the variable NEW has no value the first time it is tested at the top of the WHILE loop. More subtly, this program will incorrectly add the 99999 into the final sum. This occurs because, once in the loop, the value of NEW is first read, then summed, and only then tested for equality to 99999; thus, by the time the test is made, the 99999 has already been added into SUM.

During the first stage of the bug finding process, the program is parsed, producing an abstract representation of the program. This representation makes it easy to automatically determine the types of expressions and statements, their execution time sequence, and any

nesting of statements (i.e., statements in loops or in an IF statement). Figure 5 contains the important fragment from the parse tree produced during this stage. This stage is just the standard parsing phase of a language translator, except that all occurrences of each variable are linked together (see Figure 5). Since many bugs deal with the use (and misuse) of variables, fast access to the occurrences of variables is needed in order to facilitate subsequent bug analysis.

The next step is to annotate the parse tree with useful information about the various nodes. This information is designed to simplify and summarize parts of the tree for subsequent bug finding steps. For example, the assignment statement "SUM := SUM + NEW" would be annotated as a "running-total" assignment. In Figure 6, we show the annotations for the parse tree of Figure 5. The annotations are that (1) the WHILE loop is testing the variable NEW, (2) the READ statement is reading into the variable NEW, and (3) the assignment inside the loop assigns a running total to the variable SUM.

During the third stage of processing, the Bug Finder searches for instances of the various programming plans. This is a pattern matching process that uses the annotations in the parse tree as feature detectors against which to compare the plans. For example, the features of a running-total variable plan are, roughly speaking, (1) a variable that is continually updated by (2) a new value that is generated each time through the loop. The assignment statement "SUM := SUM + NEW" fits this description, and thus SUM is inferred to be the running-total variable. Similar arguments can be given for the other variable plans and the loop plan. In Figure 7 are three plans found in the example "buggy" program: NEW is a new-value variable, SUM is a running-total variable, and the loop is an instance of a new-value controlled running-total loop plan.

Finally, given the abstract representation, the annotation, and instantiated plans, our program searches for

Figure 5. Bug Finder Stage 1: augmented parse tree. A parse tree for a fragment of the program in Figure 3. The normal parse tree has been augmented with additional links that tie together all occurrences of a variable.

Figure 6. Bug Finder Stage 2: annotations. The same parse tree as in Figure 5, except that the nodes of the tree have been annotated with information that summarizes the actions in the various subtrees.

Figure 7. Bug Finder Stage 3: plans. Plans are identified in the program, and this information is added to the parse tree.

bugs. Bugs are detected in terms of the high-level information derived from earlier phases. One class of bugs involves variable plans that could not be fully matched (e.g., failure to initialize variable). In Figure 8, the running-total variable, SUM, has such a bug. Other bugs are caused by incorrect ordering of the operations in the loop. For example, the program in Figure 4 will incorrectly add the sentinel value 99999 into the running total; this is a classic example of the "off by one" bug.

Moreover, upon entry into the loop, NEW is undefined. A correct version of this program, as also shown in Figure 4, would READ a value into NEW before the loop and would perform the READ on NEW in the loop after the running-total update. The Bug Finder identified these errors based on the previous annotations, as indicated in Figure 8.

The process of matching plans to programs and of identifying bugs in the plans is clearly a heuristic one

Figure 8. Bug Finder Stage 4: bugs. Finally, bugs are found in the program using the plan annotations in the parse tree and the catalogue of common bugs built into the Bug Finder.

and is not guaranteed to be 100% successful. While there are techniques such as data and control flow analysis that are exhaustive, they are quite low level in comparison to the plan-based analysis. Thus, it is not clear that they would even be of much use in programs of the complexity used in introductory programming courses. While we are not adverse to adding such analysis techniques, we believe that the real power of the Bug Finder will come from building up more extensive and richer libraries of plans (Rich, 1980) and bugs.

Currently, the Bug Finder is specifically programmed to identify a particular set of bugs. In order to make the Bug Finder more flexible, we are looking into making it more table driven. That is, a better procedure would be to describe the bugs that should be searched for in a high-level language and have the computer automatically compile these descriptions into bug finding procedures. Work is underway to define such a bug description language.

PRELIMINARY RESULTS

A pilot test with the Bug Finder was carried out in an introductory programming class at the University of Massachusetts. For this test, the Bug Finder was run as a tutor and was tailored to the "rainfall problem" discussed previously (see Figure 1).

A preliminary analysis of 61 users of the Bug Finder/Tutor shows that, on the average, people ran their program six times. The minimum number of runs was 1 and the maximum was 26. The bugs with the highest frequency of occurrence were: (1) failing to test that

the input datum was greater than zero. (2) Redundantly retesting the condition that ended the loop immediately after the loop. (3) An infinite loop because the new-value variable is never changed inside the loop body. This means that the loop test will never be satisfied. This bug is particularly insidious, since the student would normally be given no useful information about what happens when the program does not execute properly. (4) Initializing the count variable to one instead of zero.

Subjectively, the students were quite pleased with the Bug Finder. Two typical comments were "It made you think about your own errors and know where to look" and "Found bugs I wasn't aware could exist. I really found it a time and grade saver." Negative comments mainly cited presentation problems related to the clumsiness of the NOS timesharing system. We also received comments such as "Make it more personal, if possible." Finally, overall class grades on the assignment that used the Bug Finder/Tutor were considerably better than the teacher had expected, surpassing the grades of both the comparable assignment of the previous semester and the previous, far easier, assignment of the current semester.

An interesting question arises with respect to using the Bug Finder as a tutor vs. using it off-line to simply analyze programs. Is the pattern of bugs for students who receive immediate feedback from the Bug Finder different from the pattern of bugs for students who do not receive the benefit of the Bug Finder's help? We would certainly expect that the persistence of bugs during the program development process to drop, since

finding semantic and pragmatic bugs can be time-consuming and can require numerous runs of the program before an adequate diagnosis is formed. As the Bug Finder is used more in both modes, we will be able to better explore this intriguing question.

CONCLUDING REMARKS

In this paper, we have surveyed methodology for exploring, via on-line protocols, the psychological aspects of computer programming by novices. We have argued for the importance of this sort of data, described how such data can be automatically analyzed and put forward some preliminary results gathered using this approach. We are quite encouraged by the project's success to date. However, this project has required significant resources. The Bug Finder and associated software (e.g., CNS122) has required 5 man-years of effort plus nontrivial sums for computer time. Thus, while there is a definite need for the collection and analysis of on-line protocol data, we caution the researcher that this enterprise requires a major investment of resources.

REFERENCE NOTE

1. Teitelbaum, T., & Reps, T. *The Cornell program synthesizer: A syntax-directed programming environment* (Tech. Rep. 80-421). Ithaca, N.Y: Cornell University, Department of Computer Science, 1980.

REFERENCES

- ADELSON, B. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 1981, 9, 422-433.
- RICH, C. *A library of plans with applications to automated analysis*. Unpublished doctoral thesis, Massachusetts Institute of Technology, 1980.
- SHNEIDERMAN, B. Exploratory experiments in programmer behavior. *International Journal of Computer and Information Sciences*, 1976, 5, 123-143.
- SOLOWAY, E., BONAR, J., WOOLF, B., BARTH, P., RUBIN, E., & EHRLICH, K. Cognition and programming: Why your students write those crazy programs. *Proceedings of the National Educational Computing Conference*, 1981.
- SOLOWAY, E., EHRLICH, K., BONAR, J., & GREENSPAN, J. What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), *Directions in human-computer interaction*. Norwood, N.J: Ablex, 1982.
- SOLOWAY, E., WOOLF, B., RUBIN, E., & BARTH, P. Meno-II: An intelligent tutoring system for novice programmers. *Proceedings of the International Joint Conference on Artificial Intelligence*, 1981.

NOTES

1. There are programming environments presently available that facilitate the entry of only syntactically correct programs (e.g., Teitelbaum & Reps, Note 1). As such systems become more widely available, we believe that syntax errors will disappear.
2. Note that the data reported under "Preliminary Results" are based on running the Bug Finder on programs that attempted to solve the "rainfall problem."