READING A PROGRAM IS LIKE READING A STORY (WELL, ALMOST)

Elliot Soloway, Kate Ehrlich, Eric Gold
Dept. of Computer Science
Yale University
New Haven, Ct. 06520

1. A Schemata-Based Theory of Program Comprehension[1]

A computer program, a mathematical proof, and an electronic circuit diagram all are (1) products of a problem solving process that required specific technical knowledge and (2) can be "executed" to obtain a specific result. However, these entities can also communicate information beyond simply the desired specific goal: in reading such "problem solving texts" and analyzing the techniques, style, comments, digressions, etc. one can gain insight into the problem itself — and even into the problem solver himself. "Stories" can also be conveyed as texts; on the other hand, they typically are not the products of technical knowledge nor are they executed for a specific result. As such, stories differ from problem solving texts. However, these two text forms are similar when both are used as a communicative vehicle. This similarity between the two text forms serves to raise an intriguing question: *Do the information representations and processing strategies that underlie the comprehension of stories also underlie the comprehension of "problem solving texts"?* In particular, can the schemata-based approach to story understanding be productively used in developing a theory of how programmers read and understand computer programs? In this brief paper, we will outline an affirmative answer to this question and describe one empirical study that supports our position. (See also [15, 16, 7, 9].)

The term we have used to express the notion of schema in the domain of computer programs is *programming plan*. Just as a schema [13, 3, 2]captures generic knowledge, a programming plan specifies the critical information that is representative of the stereotypic action sequences in programs. For example, we can identify two types of programming plans in the program in Figure 1: control flow plans and variable plans.[2] For example, the RUNNING TOTAL LOOP PLAN and the SKIP GUARD PLAN are two control flow plans in this program. The former plan repeatedly reads in some values and accumulates their total. The latter plan is also a common one: it protects the main computation of the loop from an illegal input value; should the value be input, the main processing steps are skipped over. Variable plans serve to highlight the role a variable plays in a program: just as actors take on different roles in a play, variables take on different functions in a program. For example, the COUNTER VARIABLE, Count, is used to count the number of elements being accumulated, e.g., Count:=Count+1. Similarly, the RUNNING-TOTAL VARIABLE is used to accumulate a total, e.g., Sum:=Sum+Num. While both variables are updated using an assignment statement, programmers do seem to distinguish between them on the basis of their functions [16].

2. Generating Plan-like and Unplan-like Programs

What makes a program plan-like rather than unplan-like is the way in which plans are composed in a program. In particular we have identified two rules of plan composition that can be used to systematically vary the planliness of a program. These rules are: (1) vary the typicality of the plans being composed into a program, (2) modify a typical plan in an atypical manner (usually to let the plan do "double duty"). In Figure 1, we illustrate the effect of applying these rules. The programs, which all solve the problem given in the figure, were generated using the two rules above. Programs A, B in Figure 1 reflect compositions of increasingly less typical plans. Program C reflects compositions of typical plans that have been modified in such a way as to be atypical. The heart of the problem in this figure requires a SENTINEL-CONTROLLED RUNNING TOTAL LOOP PLAN. The key issues are to add up the numbers being read in while keeping the sentinel value from being added into the total and keeping the count from also being updated. What

---

1

This work was co-sponsored in part by the Personnel and Training Research Groups, Psychological Sciences Division, Office of Naval Research and the Army Research Institute for the Behavioral and Social Sciences, under Contract No. N00014-82-K-0714, Contract Authority Identification Number, Nr 154-402. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government. This work was also sponsored in part by NSF RISE under grant number SED-81-12403.

[2]Variable plans are related to, but are richer than, the computer science notion of *abstract data types*, in that plans have more properties (e.g., relatedness, goal) than are usually associated with abstract data types.

follows is a detailed plan analysis of these programs.

- In Pascal, SENTINEL-CONTROLLED RUNNING TOTAL LOOP PLAN is most appropriately realized with a WHILE looping construct [17](Figure 1A). The sentinel is prevented from corrupting the loop in the following manner:
  - a Read of the input is positioned before the loop begins
  - if the sentinel value turns up on this first Read, it will be detected *before* the loop is executed even once; in this case processing with drop down to the IF statement.
  - if the sentinel value does not turn up on the first Read, then this legitimate value is added into the running total, Sum, and the counter, Count, is updated accordingly.
  - after these updates occur, the next value is Read in and processing returns to the top of the loop where the new value is tested; should this value be the sentinel, processing will drop down to the IF statement without further processing in the loop (i.e., without adding the sentinel into the running total).

  The COUNTER and RUNNING TOTAL VARIABLES employ the standard VARIABLE PLAN initialization and update techniques: start the value off at 0, and update appropriately. Thus, the composition of the variable plans and the loop plan is accomplished using standard techniques.
- The program in Figure 1B, however, does not use a WHILE loop, but rather a REPEAT loop. In order to protect the running total and the count from being incorrectly updated, a SKIP GUARD PLAN is used that encloses these update steps. In otherwords, there is a *causal relationship* between the LOOP PLAN and the GUARD PLAN: we need the GUARD PLAN to make up for the LOOP PLAN's inadequacy. SKIP GUARD plans are typical techniques in programming; we see one used to protect the average calculation from a divide by 0 case. Again, the COUNTER and RUNNING TOTAL VARIABLES employ the standard VARIABLE PLAN initialization and update techniques. While the composition of the variable plans and the loop plan *and* the skip guard plan is accomplished using standard techniques, it is less typical to realize a SENTINEL CONTROLLED LOOP PLAN with a REPEAT loop composed with a SKIP GUARD PLAN. This judgement of typicality is based on experience in teaching Pascal from numerous textbooks and on articles describing good programming style [17].
- While the program in Figures 1C still achieves the overall objective, it was constructed by taking standard plans and modifying them in an atypical manner. For example, the sentinel value must again be backed out of the running total variable and the counter variable. This time, however, the initialization technique of the two variable plans are modified to serve this additional function: to say the least, initializing a variable to -99999 is a very curious construction.

Does planliness effect program comprehension? One of the most important implications of a schema is that it provides a structure for comprehending and encoding information. Researchers have shown that a story is remembered better if it is more schema-like e.g. [2, 11, 5]. Similar results have been obtained for comprehension in non-story domains [6]. In the next section we will present one study in which we examined the this the issue of planliness and program comprehension using versions of the programs shown in Figure 1.

3. Empirical Evidence: A Taste

Advanced programmers (end of at least second semester of programming) were split into two groups; half were presented with program Alpha in Figure 2, while the other half were presented with the program Beta. Both groups were asked to fill in the blank line with a line of code that, in their opinion, most reasonably completes the program. Subjects were not told what problem the program was intended to solve. A version of this technique was used by Bower, Black, and Turner [3] and Kemper [10] in order to tap into the schemata people used in comprehending stories. Our hypothesis is that if programmers are using programming plans to comprehend the programs, then the expectations set up by those plans will make it easier to fill-in-the-blank in the more plan-like programs (Alpha, Figure 2). However, we suggest that it will be more difficult to comprehend the less plan-like program (Beta, Figure 2), since few expectations will be set in motion.

The results are displayed in Figure 2C. The correct answer for problem Alpha was Count := 0, while the correct answer for Beta was Count := -1. Based simply on the number of correct and incorrect answers, it was clear that program Beta elicited very different performance from that of program Alpha: there were more correct responses to Program Beta than to Program Alpha ($X^2 = 47.7$, $p < 0.001$). Moreover, it is not just that there are differences in the accuracy of the responses but also that subjects took longer to give their responses in the unplan-like program. An analysis of variance on the time to read the program and fill in the blank reveals that subjects took longer to read the unplan-like program (Beta) than to read the plan-like program (Alpha) ($F[1,91] = 4.60$, $p < 0.05$). There was no difference in reading time between correct and incorrect responses ($F[1, 91] = 3.06$, $p > 0.05$). The result that is particularly interesting is that there is an interaction between the factors of program and response: the difference in response time between correct and incorrect responses is much greater for Program Beta than for Program Alpha ($F[1, 91] = 4.50$, $p < 0.05$). That is, subjects took longer to get Beta correct than to get Alpha correct. These results lend

strong support to our claim that experienced programmers use their knowledge of plans to comprehend programs and that they will therefore take longer to comprehend unplan-like programs correctly than to comprehend plan-like programs correctly.

Interestingly, standard software engineering metrics of program complexity such as (1) lines of code or (2) a Halstead [8] metric, predict that program Beta, with fewer lines, less volume, and fewer nested structures would be easier to comprehend that program Alpha. However, given our plan-based analysis, we have argued for Alpha being the less complex --- and the experimental data cited above supports this position.

4. Concluding Remarks

In this brief summary, we have attempted to indicate the direction in which our research into program comprehension is going. We have given a brief description of how one can create plan-like and unplan-like programs, and we have described results from one experiment in which we used these programs in order to examine the use of schemata in program comprehension. These results are consistent with, but more fine-grained than, previous work on the role of schemata in technical domains in general [6, 4, 5], and programming in particular [14, 1, 12].

1. Adelson, B. "Problem Solving and the Development of Abstract Categories in Programming Languages." *Memory and Cognition 9* (1981), 422-433.

2. Bartlett, F.C.. *Remembering*. University Press, Cambridge, 1932.

3. Bower, G.H., Black, J.B., Turner, T. "Scripts in Memory for Text." *Cognitive Psychology 11* (1979), 177-220.

4. Chase, W.C. and Simon, H. "Perception in Chess." *Cognitive Psychology 4* (1973), 55-81.

5. Chiesi, H.L, Spilich, G.J. and Voss, J.F. "Acquisition of domain-related information in relation to high and low domain knowledge." *Journal of Verbal Learning and Verbal Behavior 18* (1979), 257-273.

6. deGroot, A.D.. *Thought and Choice in Chess*. Mouton and Company, Paris, 1965.

7. Ehrlich, K., Soloway, E. An Empirical Investigation of the Tacit Plan Knowledge in Programming. in Human Factors in Computer Systems , J. Thomas and M.L. Schneider (Eds.), Ablex Inc., in press.

8. Halstead, M.M.. *Elements of Software Science*. Elsevier, New York, 1977.

9. Johnson, L., Draper, S., Soloway, E. Classifying Bugs is a Tricky Business. NASA Workshop on Software Engineering, in press.

10. Kemper, S. "Filling in The Missing Links." *Journal of Verbal Learning and Verbal Behavior 21* (1982), 99-107.

11. Kintsch, W., van Dijk, T.A. "Toward a Model of Text Comprehension and Production." *Psychological Review 85* (1978), 363-394.

12. McKeithen, K.B., Reitman, J.S., Rueter, H.H., Hirtle, S.C. "Knowledge Organization and Skill Differences in Computer Programmers." *Cognitive Psychology 13* (1981), 307-325.

13. Schank, R.C. and Abelson, R.. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum Associates, Hillsdale New Jersey, 1977.

14. Shneiderman, B. "Exploratory Experiments in Programmer Behavior." *International Journal of Computer and Information Sciences 5,2* (1976), 123-143.

15. Soloway, E., Bonar, J., Ehrlich, K. . Cognitive Strategies and Looping Constructs: An Empirical Study. Communications of the ACM, in press.

16. Soloway, E., Ehrlich, K., Bonar, J., Greenspan, J. What Do Novices Know About Programming? In A. Badre, B. Shneiderman, Ed., *Directions in Human-Computer Interactions*, Ablex, Inc., 1982.

17. Wirth, N. "On the Composition of Well-Structured Programs." *ACM Computing Surveys 6*, 4 (1974).

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)

```
                        PROGRAM OrangeAlpha;
                        VAR Sum, Count, Num   INTEGER;
                            Average   REAL;
Counter Variable        BEGIN
Plan          ---------> Count := 0;
              |     ---> Sum := 0;                  Running Total Loop Plan
              |     |    Read(Num);  <-------------------
Running Total |     |    WHILE Num <> 99999 DO <------|
Variable Plan |     |      BEGIN                      |
              |     -------> Sum := Sum + Num; <------|
              -------------> Count := Count + 1;      |
                           Read(Num);       <--------
                         END                          Skip Guard Plan
                       IF Count > 0 THEN  <--------------------------
                                    BEGIN <---------------------------|
                                     Average := Sum/Count; <----------|
                                     Writeln( Average); <-------------|
                                    END <-----------------------------|
                                  ELSE  <-----------------------------|
                                    Writeln( 'no legal inputs'); <----|
                        END
```

(B)

```
PROGRAM OrangeB;
VAR Sum, Count, Num   INTEGER;
    Average   REAL;
BEGIN
    Sum := 0;
    Count := 0;
    REPEAT
        Read(Num);
        IF NUM <> 99999 THEN
                        BEGIN
                           Sum := Sum + Num;
                           Count := Count + 1;
                        END;
      UNTIL Num = 99999;
     IF Count > 0 THEN
                   BEGIN
                    Average := Sum/Count;
                    Writeln( Average);
                   END
                 ELSE
                   Writeln( 'no legal inputs');
     END
```

Running Total Controlled Running Total Loop Plan
implemented with a REPEAT Loop Plan that
realizes a Read-Process Loop Strategy
  composed with a Skip Guard Plan
  to simulate a Sentinel-Controlled Running Total Loop Plan
      using Count Variable Plan
            Running Total Variable Plan
            New Value Variable Plan
Skip Guard Plan
      using average calculation

(C)

```
PROGRAM OrangeC;
VAR Sum, Count, Num   INTEGER;
    Average   REAL;
BEGIN
    Sum := -99999;
    Count := -1;
    REPEAT
        Read(Num);
        Sum := Sum + Num;
        Count := Count + 1;
      UNTIL Num = 99999;
    IF Count > 0 THEN
                  BEGIN
                   Average := Sum/Count;
                   Writeln( Average);
                  END
                ELSE
                  Writeln( 'no legal inputs');
END
```

Running Total Controlled Running Total Loop Plan
implemented with a REPEAT Loop Plan that
realizes a Read-Process Loop Strategy
  composed with a Patch Plan
    Modify Initialization of Count Variable
    Modify Initialization of Running Total Variable
  to simulate a Sentinel-Controlled Running Total Loop Plan
      using Count Variable Plan
            Running Total Variable Plan
            New Value Variable Plan
Skip Guard Plan
      using average calculation

Figure 1: Examples of Programming Plans

(ALPHA)

```
PROGRAM OrangeAlpha;
VAR Sum, Count, Num   INTEGER;
    Average   REAL;
BEGIN
    Sum  = 0;
    ----------------------------
    |                          |;
    ----------------------------
    REPEAT
        Readln(tty); Read(tty,Num);
        IF NUM <> 99999 THEN
                        BEGIN
                           Sum  = Sum + Num;
                           Count  = Count + 1;
                        END;
      UNTIL Num = 99999;
    Average  = Sum/Count;
    Writeln(tty, Average);
END.
```

(BETA)

```
PROGRAM OrangeBeta;
VAR Sum  Count, Num   INTEGER;
    Average   REAL;
BEGIN
    Sum  = -99999;
    ----------------------------
    |                          |;
    ----------------------------
    REPEAT
        Readln(tty)  Read(tty Num);
        Sum  = Sum + Num;
        Count  = Count + 1;
      UNTIL Num = 99999;
    Average  = Sum/Count;
    Writeln(tty, Average);
END.
```

	Correct		Incorrect	
	mean time	n	mean time	n
Version Alpha	2.26	37	2.41	10
Version Beta	3.98	3	2.42	41

(the time is in minutes)

Figure 2: FIB programs