PROGRAMMERS' MENTAL MODELS OF THEIR PROGRAMMING TASKS: THE INTERACTION OF REAL-WORLD KNOWLEDGE AND PROGRAMMING KNOWLEDGE

Hank Kahney & Marc Eisenstadt
The Open University
Milton Keynes, England

INTRODUCTION

This paper describes our ongoing research into the behaviour of novice programmers. We are interested in the mental processes which occur when novices are confronted with a problem statement, and the mechanisms by which they understand the problem, design an algorithm, code it, and (if necessary) debug it. Our research is a development of earlier work on problem understanding (Hayes & Simon, 1974), models of programmers' coding processes (Brooks, 1977), and debugging (Sussman, 1975; Goldstein, 1975; Laubsch & Eisenstadt, 1981).

We investigate students attempting to write recursive inference programs using a LOGO-like database-manipulation language called SOLO (Eisenstadt, 1978; Eisenstadt, Laubsch, & Kahney, 1981) Students are presented with a prototypical problem and solution couched in everyday terms in order to simplify the explanation of recursion: "Imagine a chain of 'KISSES' relations, e.g. JOHN KISSES MARY KISSES FRED KISSES JANE, etc. A procedure called INFECT can propogate FLU all the way through the chain of KISSES relations, so we end up with JOHN HAS FLU, MARY HAS FLU, etc." The example is explained to the students in great detail, including several pages of text, diagrams, and a worked-through trace of a sample invocation of INFECT.

As one might expect, some students 'get it' (i.e. understand this simple form of tail-recursion and the notion of propogating side-effects through the data base), and some don't. The difference between those who 'get it' and those who don't can be accounted for by differences in (a) the abstractions they make from their first detailed example, and (b) the evaluation rules inherent in the mental models they use to 'run through' trial solutions.

A SAMPLE PROBLEM AND SOLUTION

We investigated students solving several recursive inference problems, including one based on a real-world example so compelling that we could be 'certain' the nature of the task was perfectly understood. Here is a concise summary of the problem:

Given a database describing objects piled up on one another as follows:

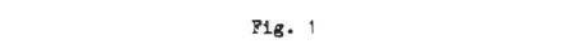

Fig. 1

write a program which simulates the effect of someone firing a very powerful pistol aimed downwards at the topmost object (SANDWICH), yielding the final database shown below:

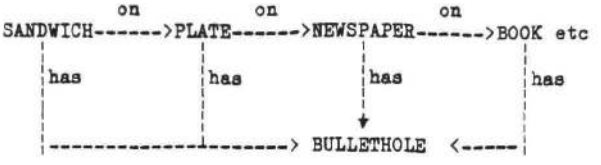

Fig. 2

As it turns out, even our 'crystal clear' example (fleshed out in considerably more detail) causes difficulty-- it appears that those students who 'get it' can cope with either 'crystal clear' or 'muddy' recursive inference problems, whereas those who don't are stuck in either case.

Fig. 3 below shows the solution eventually produced by subject S8, one of the subjects who 'got it':

```
TO SHOOT /X/
 1 NOTE /X/ HAS BULLETHOLE
 2 CHECK /X/ ON ?
  2A If present: SHOOT *; EXIT
  2B If  absent: EXIT
```

Fig. 3 S8's solution (the '*' and '?' are co-referential)

Below is a summary of the protocol of subject S8 during the course of reading and solving this problem, but before any attempt to write the code shown in Fig. 3. Problem statements are underlined. The numbers are segments from the actual protocol. It has been condensed for expository purposes in this brief paper, but captures the highlights of the protocol. A complete version is described in Kahney (1982).

"On page 80 of Units 3 to 4 we looked at a method for making a particular inference 'keep on happening'."

> 2 Is that called 'iteration'? No, 'recursion'... I think this is going to say something about what happens when you keep on applying a function...through a database

"In this option you are asked to imagine a state of the world in which there are six objects: ... this hypothetical world is highly structured: the sandwich is lying in the centre of the plate, which is sitting on the newspaper, which is lying on the book ..."

> 4 ... well you could also get out things like... sort of making inferences about 'if the sandwich is on the plate which is on the newspaper [then] the sandwich is on the newspaper'.

"A database representing this state of affairs looks like this [Fig. 1]. Now imagine someone standing beside the table with a .357 magnum pistol."

8 Well, I would expect him to shoot through all that lot then. I don't know why he wants to do it though...

S8 began with a working knowledge of recursive procedures. At successive sentences S8 set up expectations about what would come next and usually was in the position of predicting the information contained in the next sentence or two: she was always just slightly ahead of the game. S8 is apparently using a 'recursion' schema to direct her attention during the reading process to important aspects of the problem statement. The first line of the problem statement has clearly triggered off an expectation of recursion [protocol segment 2], with a concomitant expectation of some 'function' to be applied 'through a database' [segment 2]. The database structure [Fig. 1] is consistent with her expectation of a standard transitivity problem [segment 4], even though this is not the problem to be posed. Her real-world knowledge about pistols and the spatial relationship of the objects in the problem combines with her expectations about transitivity problems to yield an expectation about what the protagonist in the problem statement will do [segment 8]. This expectation does not mesh with her knowledge of human motivations and intentions [segment 8].

Figure 4 depicts our representation of S8's internalized schema for recursion. The details of the schema are derived from a variety of sources: transcription tasks, concept rating and sorting tasks, problem-solving tasks, and verbal protocols.

RECURSIVE-PROCEDURE

```
 ...
GOAL: (ForEvery x In (my applies-to) do
         (achieve (my action) x))
ACTION: (a side-effect {DEFAULT (a NOTE)})
APPLIES-TO: (a transitive-chain)
SURFACE-TEMPLATE:
   TO (name1 =(a name)) (a parameter {default: X})
   (my action)
   CHECK (a node) (a relation) (a wild-card)
    IF PRESENT: (a procedure
                  with name = name1
                  with parameter = "*");EXIT
    IF  ABSENT: EXIT
   DONE
EVALUATION-RULES:
  1) (let parameter = the startnode from
                      (my applies-to))
  2) (apply (my action) parameter)
  3) (assert `(ACHIEVED ,(my action) ,parameter))
  4) (let parameter = (GetNextNode))
  5) (ForEvery x In (GetRestOfNodes)
        (assert `(ACHIEVED ,(my action) ,x))
TRIGGERS: "keep on happening"; re-apply
```

Figure 4: S8's schema for recursion

Bearing in mind that slot-names are displayed against the left-hand margin (e.g. GOAL, ACTION, etc.), and that the function "my" is a cross-reference to a slot-filler (e.g. (my action)) we can paraphrase S8's schema for recursion as follows:

The GOAL of a recursive procedure is to perpetrate a side effect on every element of the data structure to which it is applied, i.e. a 'transitive' chain. (Knowledge about such structures is contained in S8's TRANSITIVE-CHAIN schema, not depicted here, which indicates that a collection of nodes standing in a particular relation to one another is an essential component of recursive processing-- S8 has abstracted this notion, although KISSES is not a transitive relation.) The ACTION involved is typically the application of a NOTE primitive (which performs a database 'ASSERT'). The SURFACE-TEMPLATE depicts raw SOLO code, with its own slots to be filled in during actual coding. It is based upon an exempla given in the textbook, and corresponds to rote learning of 'how to do it', rather than understanding of 'how it works'. (Subjects like S5, discussed below, have a poorer grasp of recursion and need only have a mental pointer to a place in the textbook where they can find a typica example to copy.)

'How it works' understanding is reflected primarily in the GOAL and EVALUATION-RULES slots. The GOAL slot captures the essence of the 'generator plan' used in the program-understanding plan-libraries of Waters (1978) and Laubsch & Eisenstadt (1981). The EVALUATION-RULES slot depicts S8's technique for working through a mental model of the succession of effects carried out by a body of SOLO code. The rules are clearly not sufficient to work as a SOLO interpreter, but rather depict the subject's own naive strategy for convincing herself that the code 'works'. The rules behave as follows: (1) instantiate the parameter, pretending that it's the first node in the chain (i.e. SANDWICH); (2) imagine the main action being performed on that node; (3) make a mental note that the action has been achieved; (4) see what node is next in the database, traversing the crucial 'transitive' relation; (5) make a mental note that the action is achieved on every node reachable along the 'transitive' chain.

Below we present S8's protocol corresponding to the above evaluation rules, along with the relevant rule listed in square brackets, e.g. [ER1], [ER2], etc. These protocol segments were recorded after S8 had written the program, but before she ran it.

208 TO SHOOT... X, let's say X is a SANDWICH... [ER1]

210 First of all it NOTEs in the database...X HAS BULLETHOLE [ER2, ER3]

211 It then CHECKs whether X is ON anything... [ER4]

213 X is ON PLATE so it will do that to PLATE... So that should keep doing that, PLATEs on ... something, so on and so on... [ER5]

A SECOND SOLUTION

Here is the solution eventually developed by subject S5, who didn't 'get it':

```
TO SHOOTUP /X/
 1 NOTE /X/  HAS  BULLETHOLE
 2 CHECK /X/  SHOOTS  ?
  2A If Present: SHOOTUP  *; EXIT
  2B If Absent: EXIT
```

Figure 5

Below are extracts from S5's protocol. Whereas S8 was able to develop the solution 'in her head', S5's solution evolved during code-writing:

46 I'm going to follow that example [* INFECT].

51 [Reads from INFECT example in SOLO primer]... NOTE..um, X HAS FLU... SANDWICH HAS BULLETHOLE....

54 SANDWICH ON PLATE, um....NOTE...um...

63 I've got to get the SHOOT in somewhere haven't I?

65 CHECK...X SHOOTS SANDWICH. IF PRESENT....SHOOTUP....

83 Well, I hope it will go all the way through the sequence and shoot the floor. The data base is in and I've copied that program [= INFECT] exactly.

S5 has a recursion schema which differs from that of S8 in several respects. First, S5's schema does not have a filled SURFACE-TEMPLATE slot, but rather (a) a pointer to the place in the SOLO primer where a typical recursive procedure, i.e. INFECT, is described, and (b) a method for filling the SURFACE-TEMPLATE slot by copying the INFECT program's structure and providing arguments from the current problem. Second, S5's schema has a restriction that the relationship between objects in the database must be 'active' for recursion to work. That is, from the original INFECT teaching problem with JOHN KISSES MARY KISSES FRED, S5 had abstracted the rule that a start-node has to 'do' something to a successor-node before a side-effect can be perpetrated on the successor-node. (S8, on the other hand, had abstracted 'transitivity' from previous study of the INFECT program-- neither view, of course, is perfectly correct).

For example, 'ON' is not an 'active' relationship between the objects (SANDWICH, PLATE, etc.) given in the problem statement. 'ON' is passive and thus does not 'support' S5's notion of recursion. 'SHOOTS' is an active relation, and S5 is convinced that somehow SHOOTS must be brought into the pattern-matching segment of the program in order to make the program work at all [segments 63 and 65 of S5's protocol]. This conviction precludes solution of the problem, unless careful re-analysis of the example program leads to reformulation of the rule about relationships between database objects. S5 never relinquishes her belief that an active relationship need exist between the nodes for recursion to work, and her reformulations of the program are all guided by this single important but wrong-headed principle. S5's protocol continues:

156 This one about the BULLETHOLE and this one with the KISSES are different. I need to say that the first X, the first parameter does something actively... to the second parameter. All I've got is BULLETHOLE. In the example it's got KISSES, which is an active thing.

Although S5 made several subsequent attempts to map the BULLETHOLE problem onto the INFECT framework, the point of view from which the mapping occurred never changed and no solution resulted.

CONCLUSION

Because our programming problems use real-world examples rather than abstract programming tasks, the subjects' knowledge of programming interacts with their real-world knowledge during the reading, coding, and debugging processes. We have indicated the way (often imperfect) knowledge of programming concepts pervades problem solving even in its earliest stages.

Our subjects develop schemas for recursion which are more or less 'adequate' for solving the problems we devise. This adequacy ranges from that of subject S5 (who can not solve any of the recursion problems we have devised) to that of subject S8 (who can solve many, but not all, of our recursion problems). When a problem maps onto an adequate set of schemas in a novice's store of knowledge, the novice can tackle the tasks of problem understanding, method finding, coding, and informal verification in a productive and efficient manner. When a problem is mapped to an inadequate set of schemas, the problem statement is often poorly understood, and becomes embedded in a program constructed as much from world knowledge as from the basic elements of the implementation language.

REFERENCES

Brooks, R. Towards a theory of the cognitive processes in computer programming. Int. J. Man-Machine Studies, 9, 1977.

Eisenstadt, M. Artificial intelligence project. Units 3/4 of Cognitive psychology: a third level course. Milton Keynes: Open University Press, 1978.

Eisenstadt, M., Laubsch, J., & Kahney, H. Creating pleasant programming environments for cognitive science students. Proceedings of the Third Annual Cognitive Science Society Conference, Berkeley, California, 1981.

Goldstein, I.P. Summary of MYCROFT: a system for understanding simple picture programs. Artificial Intelligence, 6,, 1975.

Hayes, J.R. & Simon, H.A. Understanding written problem instructions. In Gregg, L.W. (Ed.), Knowledge and Cognition. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1974.

Kahney, H. An in-depth study of the behaviour of novice programmers. Technical Report No. 82-9, Human Cognition Research Group, The Open University, Milton Keynes, England, 1982.

Laubsch, J. & Eisenstadt, M. Domain specific debugging aids for novice programmers. Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81), Vancouver, B.C. Canada, 1981.

Sussman, G.J. A computer model of skill acquisition. New York: American Elsevier,1975.

Waters, R.C. A method for analyzing loop programs. IEEE Transactions on Software Engineering, SE-5:3, 1979.