A MODEL OF HUMAN COGNITIVE BEHAVIOR IN WRITING CODE FOR COMPUTER PROGRAMS

Ruven Brooke
Department of Information and Computer Science
University of California - Irvine
Irvine, CA 92664
U.S.A.

Abstract

A theory of human cognitive processes in writing code for computer programs is presented which views behavior in terms of three processes: understanding, planning, and coding. Using this theory, a model of the coding process has been created which reproduces the code generation behavior of an experienced, human programmer working on a set of 2 3 FORTRAN problems. The model is stated as a computer program organized as a production system.

A Theory of Human Computer Programming Behavior

Programming is an activity which is engaged in, in one form or another, by more than a million people (Boehm, 1972). Surprisingly, research on human behavior in programming is very sparse and consists mainly of measurements of programmer performance. The research presented here takes a different approach than these previous studies. Instead of attempting to measure individual variables associated with programmer behavior, it presents a theory of the program writing process. The theory uses a set of ideas developed by Allen Newell (19 74) [1] and lies within the framework of the information processing approach to human problem solving (Newell & Simon, 1972).

Before presenting the theory itself, a necessary prerequisite is definition of the class of tasks for which the theory is intended. Though computer programming has been spoken of as though it were one task. In fact, a number of different tasks are included under this heading; they include, among many others, writing specifications for programs, writing programs given a set of specifications, debugging programs that another programmer has written, and writing documentation for programs. The particular task that has been selected for the focus of this theory is one In which the programmer is given a description of a function, or set of functions, which the program is to perform. The programmer must then find a method to perform the function and implement the method In the programming language. As a working situation, it is one which occurs by Itself frequently In scientific and educational programming environments and as a part of almost every other programming task. Limitations on research time and resources prevent inclusion of situations in which different parts of the task are performed by different people or of situations involving debugging behavior, though there are no obvious limitations of the theory which prevent doing so in the future.

Structure of the Theory

The theory consists of three basic processes: understanding, planning, and coding, though the work presented here focuses only on coding. Each of these processes occurs one or more times in every instance of the programming task. The operations performed by each of these processes and the way in which they invoke each other are described in the following section.

Understanding

When a problem-solver is presented with a problem he has a variety of sources of Information, both internal and external, available about it; these include his general "world" knowledge about the general type of problem at hand, reference works such as programming language manuals, and last, but not least, written or spoken problem directions - Before he can actually start work on the problem, he must use these Information sources to build representations of the basic elements that the problem deals with and of their properties. Specifically, he must have representations of the initial state of the problem, the desired final state or goal, and one or more operations which he can apply, appropriately, to begin the transformation on the initial state. The process of building these representations is referred to as "understanding" in this theory.

Planning

When the building of representations that composes the understanding process is complete, the second of the three processes, planning, is invoked. The type of plan produced by it can best be described as a method for solving the programmlng problem; it consists of specifications of the way in which information from the real world is to be represented within the program and of the operations to be performed on these representations in order to achieve the desired effects of the program. These methods are used as schemas, or outlines, to guide the writing of the actual code in much the same way as plans are used to guide solutions of logic problems by GPS (Newell and Simon, 1972). The theory asserts, based on a sufficiency argument, that such a plan exists for nearly every programming problem which fits within the basic task definition.

Plans are expressed in a functional language of the sort investigated by Newell and Freeman (1971); functions specified in the language invoke structures which, in turn, require other functions. This type of behavior may be characteristic of the whole class of design problems.

Analogous concepts are the plans used In the planning version of GPS for the logic task (Newell & Simon, p. 428, 1972) and the language accepted by the Functional Description Compiler routine of the Heuristic Compiler (Simon, 1972).

Planning does not take place as a single operation; instead, a process of step-wise refinement takes place in which step makes part of the plan more detailed. The terminating condition for the refinement is that some (reasonably large) part of the plan is sufficiently detailed so that the programmer feels that he knows how to translate it into code, even though all of the details of the code are still unknown. At that point the final process in the writing of programs, coding, takes over. The coding process operates on a piece or part of a plan until either code is produced or some criterion is met which causes the coding process to report failure; when failure occurs information is passed back to the planning process which again attempts to produce a codeable plan.

Coding

The third of the three processes in the theory is coding. For human programmers, the basic cycle for the generation of code consists of using the plan to select and write a piece of code, assigning an effect or action to the code that has been written, and comparing the effect, or action, to the stipulations of the plan. The results of this comparison are used to select and write more code or to change the code that has been written; in turn, an effect is assigned to this new code which is compared to the plan. This cycle continues until the cumulative effect of the code meets the requirements of the plan or until some condition, such as effort expenditure, is met which indicates that the piece of plan is not codeable.

The effects that are assigned to code are based on the differentiations among the data that the program must actually make in order to accomplish its purpose. Consider as an example a program for printing all the odd numbers in a set of integers. The program must differentiate between odd and even numbers in order to perform this task. An effect that could be assigned to a line of code in this program might be "if the number is odd, this branches to statement 50," a statement which uses the information about the odd-even distinction. As more lines of code are written, their effects are accumulated in a manner which also makes use of these differentiations; thus, the effects,"this loops through all the numbers" and "if the number is odd, branch to statement 50," might be combined to give "this tests each number to see if it's odd." When effects of this type are assigned to whole segments to code, the result is that the code is executed with symbols such as "odd number" replacing the real data; hence, the whole process has been named "symbolic execution."

These effects are expressed in a functional language. It resembles closely the language used to express plans, with the major difference, that the actions stated are ones that have actually been achieved rather than ones which are intended. In protocols of programming behavior, this distinction between planning statements of intent and coding statements of effect appears clearly enough so that it may be used to identify which of the two processes is taking place at a given point.

An example of a complete symbolic execution cycle for the problem just mentioned might start with the plan element, "test each number to see if it's odd." For a FORTRAN program, the programmer would begin by writing DO 10 1-1,100 and assigning it the effect, "this loops through all the numbers." Given this effect and the plan, the programmer might next write IF(L(I)/2*2 .NE. L(I)) GO TO 20 and then assign it the effect, "this tests whether it's odd and goes to 20 if it is." Finally, after closing the DO loop by writing 10 CONTINUE, the programmer would summarize the effect of all three lines as "this loops through all the numbers and tests each one to see If it's odd." Since this matches the plan element, program writing would proceed to the next plan element.

An alternative possibility in this example illustrates another aspect of symbolic execution. Suppose the programmer had known only a test for even parity. Since it was the only parity test available, he might have written IF(L(I)/2*2 .EQ. L(I)) and assigned to it the effect, "this tests whether it's even." Noting that the plan requires the opposite effect, he would then alter ".EQ." to ".NE." to obtain the test for odd parity. The general principle of which this is an example is that confronted with erroneous or inappropriate code, the symbolic execution process attempts to patch or modify it to obtain the desired effect. This patching or modifying behavior is one of the main characteristics that distinguish symbolic execution from the sort of goal tree building and backtracking behavior seen historically in programs such as the Logic Theorist (Newell and Simon, 1972) and,more recently, In systems such as PLANNER (Hewitt, 1971). These systems rely heavily for problem solving power on the ability to backtrack to a previous, successful position: backtracking is preferred over the creation of new subgoals. In symbolic execution, on the other hand, attempting to modify or add on to what had already been done takes precedence, and backtracking is an Infrequent event.

When backtracking does take place, It may occur at several levels. In addition to attempting to code the plan element in an alternative way, the program writer may decide that the plan is at fault. When this happens, a return is made to the planning process, and an attempt is made to find a plan, or piece of plan, which is easier to code. This new attempt in planning may even require a return to the understanding process to reinterpret the problem. If the understanding process is considered to be a "top" level process and coding process a "bottom" level one, then this ability to return to the planning and understanding process represents a "bottom-up" process, and both bottom-up and top-down processes take place in programming.

A Model of Coding

With this theory as a basis, a more detailed model of the coding process has been constructed. This theory is described completely in Brooks (1975) and is summarized here. The model is intended to reproduce certain characteristics of the behavior seen in protocols of a programmer "thinking aloud" while writing a series of short programs in FORTRAN. These characteristics include the order and sire of unit for code generation. The model is stated as computer program in a dialect of the LISP programming language. The structure of the model is based on the structure for human problem-solving systems presented and defended by Newell and Simon (1972). It consists of a short-term memory (STM) and a long-term memory (LTM); the LTM consists of a production system and two other structures, MEANINGS and CODE.

STM is presumed to have a fixed capacity of a small number (less than *20)* of chunks or symbols, each of which can be used to access information of arbitrary size and complexity in LTM. The slots are ordered so that introduction of new items at one end causes old ones to be lost off the other, a process analagous to one type of human forgetting behavior.

In addition to the introduction of new items two other processes, Item modification and rehearsal, alter the contents of STM. Item modification consists of updating or adding to the information in an item already in STM, as contrasted with the addition of an item containing entirely new information. Rehearsal takes an item from the middle or end of STM and places It at the beginning so that it will not be bumped off the end and lost.

The most important structure in LTM is the production system. A production system consists of sets of pairs of conditions and actions to be performed when the conditions are met. An appropriate decision rule is employed to insure that only one aet of actions are performed at a time. Executing the actions results in some change in the state of the world so that aa the system operates different conditions are met and different actions are invoked. None of the actions involve explicit branching; rather, all control is accomplished through differences in the meeting of conditions and the execution of associated actions.

The theory asserts that a production system is the main internal control mechanism for determining the course of problem solving and the main knowledge structure. An extensive defense of the suitability of this particular control structure for modeling human behavior is given in Newell and Simon (1972, p. 804). The conditions for the production system in this theory are the presence or absence of certain items in STM. The conditions can describe items uniquely or they can be stated in terms of general classes of items. Disjunctive or conjunctive specification of combinations of Items are also possible. An example of a condition is:

> "any item which contains the word, PLAN-ELEMENT, and the unique item which is a pointer to MEANINGS."

If STM contains items which meet that specification, then the actions associated with it would be invoked. An example of a set of actions might be:

> "Rehearse the Item which contains PLAN-ELEMENT and the Item which Is a pointer to MEANINGS. Replace PLAN-ELEMENT in the first item by OLD-PLAN-ELEMENT."

Other Long-term Memory Structures

In addition to the production system the model makes use of two other long-term memory structures. The first of the two is used for storage of the body of Information about the program that gets built up as the writing of code proceeds. Some of this information is contained in the code Itself, but much of It, such as the meanings of variables and labels and the effects of pieces of code, cannot be retrieved from the written code alone and is used over much too long a time period for It to remain in STM, at least in an unencoded form. Because of the problem of dynamically adding information to a production system, in the program this information is contained in a structure outside the production system called MEANINGS. MEANINGS Is organized as a set of attribute-value pairs, one set for each variable or expression. Examples of the attributes include the TYPE of the expression - pointer, label, array, *etc.* - and the NAME that is actually used for It in the FORTRAN program. Addition of information to MEANINGS and retrieval from it are accomplished by two special functions, NEWMEANING and GETMEANING, which are called by the production system.

The third major LTM knowledge structure in addition to the productions and MEANINGS, CODE, is actually information about how to access an external memory, the code that the programmer has already written. It Is quite likely that very little of the actual code remains sccessible in LTM once it has been written out on paper; when the subject In this study wanted to rewrite or reuste pieces of code, longer than a line or so, that he had already written, he was almost never able to recall them directly from memory. Any use, modification or correction to code which has been written must therefore retrieve the code from the paper external memory; and the LTM must contain the information necessary to perform the retrieval. The CODE structure in LTM contslns this Information. Since no experimental data were available on how subjects actually organize this information in memory, a simplified structure for CODE has been assumed; it is always searched linearly, most recent code first.

Knowledge Representations in the Model

The preceding section describes the basic knowledge structures of the model. The following section describes the way knowledge is represented within these structures.

The Plan

According to this theory of programming, a plan consists of a sequence of operations which must be performed in order to achieve the desired effect of the program. In the model, a plan is represented basically as a single list, each item of which is a single operation of the plan. A production places these items one *at* a time into

STM for coding. In some situations It Is necessary to indicate that a group of these Items is to be performed together; examples might be to show that all the Items In the group belong Inside the same loop or that they are part of the same branch of a conditional. For this purpose, special marker elements, somewhat like BEGIN and END in ALGOL, are provided which may be placed before and after sets of Items to Indicate that they belong together in a group.

In the protocols a single, functional language is used to talk about both plans and the effects of pieces of code. This Is reflected within the model by using a single notational system to represent both. The general form that plan elements expressed in this notation take Is:

<function to be performed> <operands>

A few examples of actual plan elements, with explanations, are given below:

a. ORDER LIST-OF-NUMBERS

"Order the list of numbers."

b. BRANCH-IF (EVEN-PARITY; (LIST-OF-NUMBERS; POINTER(NEXT-ODD)));
GOTO LOOP-END

"If the element in list of numbers which is pointed to by the pointer for the next odd number is even, go to the end of the loop"

c. FIND-EXISTENCE ((FIRST POSITIVE); LIST-OF-NUMBERS) BEGIN!
(OTHERWISE)
SET CORRESPONDING-ELEMENT(AUXILLARY ARRAY); VARIABLE (LOOP-INDEX)
END!
END! (FIND-EXISTENCE-LOOP-THROUGH)

"Loop through the list of numbers until the first positive is found. If a number is not positive, then set the corresponding element of the auxillary array to the value of the loop index."

The elements beginning with BEGIN! and END! are the special marker elements mentioned previously.

A final comment about this notation as applied to plans is that it makes no distinction between plan elements which lead to the generation of actual program code; for example, "set the pointer equal to the index of the first odd number found," and those which only result in the establishment of data representations, such as, "create a pointer to keep track of the location of the first odd number."

Templates

Since the plan itself is presumed to be language-independent, the information about the syntax and semantics of the language in which the code is actually written must be contained in the production system. For syntactic information this is done by means of structures called coding templates which are formally equivalent to a Backus-Normal form definition of the language, using very high-level primitives and very few recursion slots. Each template consists of a small segment of code - at most 3 or 4 lines - specified as a mixture of three types of information: actual code elements, descriptions or specifications of code elements which are to be inserted in the code, and parameter slots which can be replaced by descriptions or specifications at the point when the template is actually used. The descriptions of code elements may. In fact, be another template. An example of a template is:

DO "label for (parameter 1) loop" "variable for loop index"
= "begins at (parameter 2) loop", "ends at (parameter 3)"

Verification of the Model

The model was verified using a data base consisting of protocols of the behavior of an experienced FORTRAN programmer writing a set of 23 short programs. The problems all involved manipulations on an array, L, which contained 100 random numbers. An auxiliary array, M, was used to indicate certain things about the operations which had been performed L. A sample problem from this set is:

REARRANCE THE ARRAY SO THAT ALL ODD NUMBFS ARK AT THE BKCIMNINC.
PLACE ONES IN THE CORRESPONDING POSITIONS IN M.

While working on the problems, the subject had available both paper and pencil,and a terminal connected to an interactive computer system. For each program he was given a problem description and the name of a file containing code to read in and write out the random numbers. His instructions were to write and debug the program and to "talk aloud" while doing so. A video tape recorder was used to record the subjects behavior.

Transcriptions of the program-writing portions of these protocols formed the basic data for the verification. An example of part of one of these transcriptions is:

```
S2:REARRANGE THE ARRAY
S3:SØ THAT ALL NUMBERS
S4:ARE AT THE BEGINNING
S5:PLACE- ON, SO THAT ALL ODD NUMBERS ARE AT THE BEGINNING
S6:PLACE ONES IN THE CORRESPONDING POSITIONS IN M
S7:ALRIGHT, THAT SHOULD BE EASY ENOUGH
S8:GO THROUGH THE ARRAY
S9:DETERMINE IF A NUMBER'S ODD OR NOT
S10:HAVE A POINTER TO THE LAST PLACE WHERE THERE'S NOT AN
S11:TO- IF IT'S RIGHT AT THE BEGINNING THEN YOU KNOW HOW FAR YOU HAVE
S12:ON ODD AND HOW FAR YOU HAVE ON EVEN
S13:SO,POINTER ONE
S14:POINTER ONE IS A POINTER TO
A15:[WRITES PTR1]
S16:IT'S A POINTER FOR NEXT ODD
S17:NEXT ODD
S18:AND THE OTHER ONE IS JUST GOING TO GO THROUGH THE ARRAY SO
S19:I'LL JUST WRITE THIS NEXT ODD
S2Ø:START OUT AND
A21:[WRITES NEXTODD=1]
S22:POSITION ONE LET'S JUST SAY
S23:DO 1Ø I EQUAL
S24:LET'S MAKE THIS A 2Ø
A25:[WRITES DO 2Ø I=1,1ØØ]
S26:I EQUALS ONE TO 1ØØ
S27:IF THE THING IS ODD -IF
```

The code that the subject wrote in the problem from which this transcription was taken was:

```
   NEXTODD=Ø
   DO 1Ø I=1,1ØØ
   IF (L(I)/2*2.EQ.L(I))GOTO 1Ø
   NEXTODD=NEXTODD+1
   K=L(I)
   L(I)=L(NEXTODD)
   L(NEXTODD)=K
1Ø CONTINUE
```

From the 38 segments which were identified as coding behavior in the 23 protocols, four were selected for modeling by the program.

A number of criteria were used for selection, including the extent to which they were representative of other coding segments and their suitability for effectively testing the theory. For each of these segments input to the program consisted of a statement of the plan derived from the protocol and stated in the notation described previously. (For a complete description of how this was accomplished, see Brooks (1975).) For the segment consisting of lines 15-60 of the protocol just presented, part of this plan appeared as:

```
1.  (CREATE (POINTER (NEXT ODD))  (BEGINNING (LIST OF NUMBERS)))
2.  (LOOP-THROUGH(LIST OF NUMBERS))
3.  (IF((EVEN PARITY)
        (ARRAY-ELEMENT(LIST OF NUMBERS))
        (VARIABLE(LOOP-INDEX))))
    (GOTO LOOP-END))
4.  (BEGIN!(NOT-EVEN-PARITY))
```

Using this segment or protocol as an example, it is also possible to see how the plans were derived from the protocol. In this case, planning began very soon after the subject received the problem description; It takes place in lines 8 through 12 of the protocol, about 30 seconds after the subject received the problem description. Since this is the only identifiable planning behavior seen in the protocol, it is assumed that planning was completed in this segment and that this same plan was used without modification throughout the writing of the entire program. This segment shows that the plan consists, in part, of looping through the array, testing each number, and keeping a pointer to the last position at which a non-odd occurs. The subject's comments as he is writing code in lines 16 and 16 Indicate that the plan actually consists of keeping two pointers, one for the position at which the next odd is to be placed and one which goes through the array pointing at the next element to be tested. Finally, from his comments in lines 52-59 (not shown) as he is checking over the program and from the code he actually writes, it may be inferred that, once he has found an odd number, he intends to Increment the pointer to the next odd, swap the odd number with the element pointed to by the pointer to the next odd, and then set the corresponding element to 1*

The production system takes the steps of this plan one by one, places them into STM and attempts to convert them into code. A sample of a trace of the production system on the plan Just given is:

```
Cycle 1Ø.
Elements present in STM:
1.  (NEW-CODE*ANY*)
2.  (PLAN-ELEMENT*REST*)
3.  (CODE)
4.  (MEANINGS)
Action performed:
NEW-CODE-3
```

The "Elements Present In STM" are the elements which served as the invoking conditions for the production that fired off on the 10th cycle of the production system. NEW-CODE-3 is the action part of the production; It consists of changes to be made to the contents of STM and the other memory structures in the system.

The production system models behavior in two respects. First, It generates essentially the same code, including errors, as does the subject; the only differences lie in the areas of variable names and some slight differences in the order of code generation.

Second, the production system uses knowledge structures which correspond in size and general organization to those used by the subject. An example is the way in which IF statements are generated. In the production system this code is generated with 2 templates, one for the basic IF statement and GOTO, and a second one, invoked from within the first, for the test inside the IF statement. Corresponding behavior in the protocol indicates that the subject also divides this knowledge into the same two units.

Assertions Made by the Model

The mode makes 3 general assertions about coding behavior:

1. Coding knowledge is organized as a very large number of unique plan elements, each of which has associated with it the specific Information for translating it Into code.

2. This translation is accomplished by a symbolic execution process In which, as each line of code is laid down, a recognition process assigns It an effect.

3. The Information about programming language syntax used in laying down the code la organized as a collection of small pieces of knowledge, each of which specify how to write code for a desired action or operation.

The major support for these assertions comes from the correspondence between the behavior of the model and the behavior of the subject in the protocols. Additional support for each of the assertions comes from the following analyses:

1. The aasertlon about the structure of coding knowledge was further verified by estimating the number of new productions that would be necessary to code the plans for four additional segments. It was found that additional productions were necessary for each new plan and segment, and that the nunber of additional productions necessary did not decline as the total cumulative size of the production set grew. This is a strong indication of the large amount of knowledge specific to each plan element that makes up a programmer's knowledge of how to create code.

2. The assertion about the role of code creation in symbolic execution was further supported by asking an experienced FORTRAN programmer to Judge whether symbolic execution was visible in the protocols. He was able to find it in 33 of 35 segments of coding behavior, indicating that symbolic execution is a ubiquitous feature of this set of protocols.

3. To add support to the assertion about the representation of syntactic knowledge, an estimate was made of the total number of templates necessary to represent completely the subject's knowledge of FORTRAN syntax. The importance of this number is that it indicateswhether the template is an appropriately sized unit for modeling syntax knowledge. When an enumeration of the syntax constructions, not observed in the protocols, was used to conservatively estimate the total number of templates needed the result was 54, suggesting an absolute upper bound of no more than 150. This figure is of approximately the correct magnitude, indicating that the template is of an appropriate size to represent syntax knowledge.

Implications for Artificial Intelligence

Using Productions Systems to Model Parallel Processes

One of the central features of this model is the use of a production system as both the central control mechanisms and the primry knowledge structure. An important question is whether this was an appropriate choice. A primary characteristic of a production system for modeling behavior is that, at a given point, the selection of the next piece of behavior is made in parallel from all the possible alternatives. Any sequential dependencies seen in the behavior of a production system is the result of a specific implemenatlon and is not an inherent characteristic of the control structure itself. A structure that was essentially different from a production system would select behavior via a sequence of decisions that was an inherent part of the control mechanism structure. To argue that production systems are a particularly good choice for representing coding behavior requires that there be some aspect of the coding process which cannot be easily represented in serial fashion.

In this case, a strong argument for essential parallelism can be made from the retrieval of knowledge about the association between plan elements and code. One of the main findings of this study has been that a programmer has a large body of knowledge about how to code particular plan elements. Since this body is so large, serial processes in searching it ought to reveal themselves by extremely long retrieval times for information about how to code most plan elements, perhaps on the order of several minutes. Additionally, the protocols ought to contain some evidence of sequential elimination of unwanted information until the correct solution is found. In this set of protocols, once the subject has a plan, coding of it seems to start almost immediately without any utterances which would indicate that the subject has to expend effort to figure out how to begin. Additionally, while there are several Instances in which the subject considers alternative methods of doing coding, there is no evidence of a fixed, sequential elimination of unwanted alternatives.

This evidence strongly suggests that the search for coding information is made in a parallel manner. While it is true that it is possible to map a parallel process onto a serial model, both evaluation and explication of the model can be accomplished more effectively when the correspondence between the model and the process is a clear one. Given the parallel nature of the search process in this case, representing it with a production system is particularly appropriate.

Implications for the Use of Backtracking

One of the most common ways to organize a problem solving system is as a backtracking subgoaler. Systems with this organization attempt to solve problems by reducing them to a set of subproblems, whose solution imply the solution to the Initial problem. When the system falls at a subproblem which it has attempted to solve, it returns or backtracks to some prior, successful state. Systems of this type vary considerably along such dimensions as the strategy used to generate and select subgoals, the amount and kind of information retained from failures, and the point to which return from failure occurs (Nilsson, 1971; Newell & Simon, 1972).

While the model presented here is not organized as a backtracking subgoaler, it is of interest to inquire what role backtracking plays in coding behavior. In its purest form, backtracking in programming would consist of completely abandoning some piece of code by erasing it or crossing it out and beginning again at some earlier point, up to which the code was known to be correct. In these protocols, this type of behavior occurred in only one problem. A far more common occurrence, taking place in 21 to 23 of the protocols, was that the unsatisfactory code is modified by insertion of lines, crossing out, changing names, etc., until it is corrected. In the production system this behavior is accomplished by productions which hsve as their invoking conditions the effects that have been assigned to the unsatisfactory code and which, as their effects, modify the existing code.

The point to be emphasized about this behavior is that, when a failure to generate correct code occurs, as much of the old solution attempt as possible is saved and reused. This is in strong distinction to systems such as GPS (Ernst & Newell, 1969) and PLANNER (Hewitt,1972) which, when failure occurs, discard the old solution attempt entirely. The contrasting way in which the subject and the model in this study behave has the effect reducing the number of goals attempted at any one level since the information obtained from attempting one goal is available in the form of written code for use by successive goal attempts. The goal "tree" thus becomes more of a straight line. Generalizing from this, human computer programming suggests that a paradigm of "patch and move forward" is better suited to some tasks than the wide-spread "backtrack and subgoal."

Directions for Further Work

The model presented here covers only the coding process and is, strlckly speaking, applicable only to this one subject. Future work should attempt to broaden it both in terms of the processes covered and generality for other individuals. Some of the research questions ought to be:

1. How are plans created by human programmers? In particular, is the creation of new plans an active problem solving process or does it involve primarily retrieval of stored plan Information?

2. To what extent and in what ways does the programmer's knowledge of the programming language he is using affect the plans he uses?

3. Is the generation of code by symbolic execution an invariant across programmers and situations or are there other methods of coding that human programmers use.

Not only do answers to these questions represent intriguing problems in their right, but the answers to them may prove useful in applied work on improving software technology.

Footnotes

[1] The elements of the Newell theory that are used here are:

1. Development of plans by heuristic search consisting of successive functional elaboration in which functional specifications invoke structures which, in turn, require further functions.

2. Generation of code by a symbolic execution process in which first, code is laid down and then consequences are generated from it. The consequence generation produces a large number of subproblems.

3. Solution of these subproblems by a recognition process. Together with symbolic execution, this implies goal control dependent on the problem structure, rather than via a goal stack.

Bibliography

1. Boehm, B. W. Software and Its Impact: A Quantitative Assessment. RAND Corp. 1972.

2. Brooks, R. A Model of Human Cognitive Processes in Writing Code for Computer Programs. Doctoral dissertation. Psychology Dept. Carnegie-Mellon University, 1975.

3. Ernst, C. W. & Newell, A. *GPS: A Case Study in Generality and Problem Solving,* Academic Press, New York, 1969.

A. Freeman, P. & Newell, A. A Model for Functional Reasoning in Design. *Proc. International Joint Conference on Artificial Intelligence,* 1971.

5. Hewitt, C. Description and Theoretical Analysis of PLANNER. Doctoral Dissertation, Massachusetts Institute of Technology, 1971.

6. Newell, A. Notes on the Psychology of Programming. Computer Science Department Carnegie-Mellon University (forthcoming).

7. Newell, A. & Simon, H. A. *Human Problem Solving,* Prentice-Hall, New York, 1972.

8. Nilason, N. J. *Problem Solving Methods in Artificial Intelligence,* McGraw-Hill, New York, 1971.

9. Simon, H. A. The Heuristic Compiler, in Simon, H. A. & Sikloesy, L. (Eds.) *Representation and Meaning,* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.