Towards a Theory of the Cognitive Processes in Computer Programming

by

Ruven Brooks

Department of Information and Computer Science
University of California Irvine
Irvine, CA 92717

Technical Report #100

Abstract

While only in the past ten years have large numbers of people been engaged in computer programming, a small body of studies on this activity have already been accumulated These studies are, however, largely atheoretical. The work described here has as its goal the creation of an information processing theory sufficient to describe the findings of these studies. The theory postulates understanding, method-finding, and coding processes in writing programs, and presents an explicit model for the coding process.

In the past thirty years, computers have moved from academic curiosities to essential tools in business, government, and education. With the rapid growth in the numbers of machines has come an equally rapid expansion in the numbers of personnel working with the machines, particularly in the area of software generation and programming. One estimate is that, in the U.S., more than a million people are involved in some aspect of the programming process (Boehm, 1972).

The large number of people engaged in this work, as well as the complexity of the task itself should make programming behavior of considerable interest to cognitive psychology. Indeed, though significant quantities of programmers have been available for only the past 10 years, researchers have been quick to turn their attention to cognitive aspects of programming, and a small, but growing body of studies now exists. As is reasonable to expect in a newly developed field, most of the studies are atheoretical. They attempt to establish the validity of various parameters for describing programming behavior, rather than attempting to specify underlying processes which determine these parameters. The intent of this paper is to begin to develop a theoretical framework for the study of some of the cognitive processes involved in one kind of computer programming. The goal of this effort is not to immediately derive testable consequences from the theory, but, rather, to demonstrate the sufficiency of the theory, or of direct derivations from it, for explaining observed programming behavior. In this respect, the theory very much shares the philosophy behind the work of Winograd(1972) and of Anderson and Bower (1973).

Review of Existing Work

A useful starting point for presentation of this theory is a review of the existing body of studies, with the aim of identifying some of the more characteristic features of programming behavior. Essentially, these studies can be divided into three groups. The oldest of these grew out of an early controversy about the relative merits of interactive versus batch systems. (Grant & Sackman, 1967; Smith, 1967; Schatzoff, Tsao, and Wiig, 1967; Gold, 1968). While changing economics have rendered the main thrust of these studies moot, they did succeed in establishing one important aspect of programming behavior. Even among programmers of very similar experience levels, differences of as much as 100 to 1 were found across programmers in the time taken to write a given program. Additionally, across problems constructed to be of similar difficulty, an individual programmer often displayed a six-fold difference in writng time.

A second group of studies has been addressed to features and error characteristics of particular programming language(Rubey,1968; Knuth,1971; Litecky and Davis, 1976). While these studies have not been intended to provide information on cognitive aspects of programming, they have produced the important finding that errors or difficulties in programming are not random occurences, produced by random occurrences in the programmer´s environment. Instead, they are clearly linked to specific features or properties of programming languages.

A third group of studies shares the common property of attempting to specify factors or characteristics of cognitive aspects of programming, other than strictly language dependent ones. Among the earlier of these is Young´s (1974) study of error rates in programming which revealed, surprisingly, that experienced programmers initially make about the same number of errors as beginning programmers, but are able to find their errors faster. The work of Sime, Green, & Guest (1973) is aimed at establishing links between programming behavior and psycholinguistic theory; using one such link, it successfully predicts a performance difference in the use of two different programming language constructs. Finally, the work of Boies and Gould (1974), Gould and Drongowski (1974) and Weissman (1975) all seek to establish the validity of an experimental methodology for studying various factors in programming behavior.

In addition to experimental research on programming, a large body of informal data exists in work on programming practices and techniques, often discussed under the heading, "software engineering." While much of this work is prescriptive rather than observational or experimental, it does suggest some important characteristics of programming behavior not yet studied experimentally. Perhaps one of the most important of these comes from the work on producing well-structured programs (Dijkstra,1972 ; Yourdon,1975). While most of this work is very controversial and there is, as yet, no universal agreement on goals or methods, much of it seems based on the assumption that the "cognitive load" (Bruner, Goodnow, & Austin, 1956) involved in understanding how a program should function will be an important determinant of how easy the program is to write, debug or modify. The proposed methods for reducing this load are essentially aimed at constraining programmers to organize programs heirarchically and modularly in such a manner that an operation at any one level can be broken down into a small number of simpler operations. The aim of this process is to reduce the number of units of information which are necessary to understand any given piece of program, and, thereby, to minimize the cognitive load.

The need for such tactics indicates that an important element in determining how easy a program will be to write or understand is the conceptual organization imposed on it by the programmer or reader. Thus, a programmer may find a program easier to write if he can break it down into an input routine, a sort routine, and an output routine than if he views it as 57 distinct lines of code. If this is true, then it suggests that the size and kind of conceptual units that a programmer has available will be an important determinant of his programming behavior.

A Theory of Programming Behavior

The theory of programming that will be presented here is in the context of the problem-solving theory of Newell and Simon (1972), and shares their view of problem solvers as information processing systems. Like their theory, this theory of programming is oriented towards explaining the behaviors seen in transcriptions, called protocols, of the verbal behavior of subjects asked to "talk aloud" while performing programming tasks. It also shares their bias towards modeling and predicting the specific bhavior of individuals, rather than aggregates.

Though computer programming has been thus far treated as a single activity, in many situations a variety of tasks, such as writing specifications of programs or implementing a set of specifications are included under this heading. For the focus of the theory, one particular type of task has been selected. In this type of task, a programmer is given a description of the input data and of the processing that is to be performed on it. The programmer must find an algorithm, including the selection of internal representations for the data, and implement the algorithm in a programming language. As a working situation, it is one which occurs by itself frequently only in scientific and educational environments, but it is often a sub-part of other programming tasks in commercial situations.

The theory hypothesizes that three distinct sorts of behavior, understanding, method-finding, and coding, are involved in performing this programming task. Understanding behavior is defined to be that by which the programmer acquires knowledge of the basic elements of the problem. These include the objects with which the problem is concerned, their properties and relations, the initial and final state of the objects, and the operations available for going from the initial to final states. Behavior such as reading or listening to directions, asking questions about conditions or limitations of the problem, etc. are considered to be part of understanding behavior.

In a protocol, the following kinds of statements can be classified as understanding behavior:

1. Reading the directions or problem statement.

2. Questions to the experimenter about problem interpretation.

3. References to knowledge other than that about programming, if used to determine limits or conditions of the problem. An example would be a question about the method of rotation to be used in a factor analysis program.

The underlying process in this behavior is presumed to be close to that described by Hayes and Simon (1974) for understanding written task directions. In their theory, phases of extracting of information from external sources, such as the written directions, alternate with phases of internal representation building in which new information gets incorporated. A similar process is hypothesized to underlie understanding of the problem in writing a program. Repeated passes are made across the written directions or through other information sources. Initial passes concentrate on establishing the basic objects of the problem, while later ones are concerned with adding details about the properties and relations of the objects and about their initial and final states.

The second type of behavior hypothesized to occur in programming is method-finding. A method is a plan or outline of the program to be constructed, similar in function to an architectural plan or blueprint (as versus a plan of actions to be taken). It consists of specifications of the way in which information from the real world is to be represented in the program (data structures) and of the operations to be performed on these representations to achieve the desired effect of the program (algorithms). Organization of these methods is heirarchical, with smaller methods serving as pieces of larger ones.

An important characteristic of methods is the extent to which they are independent of specific programming languages. Possibilities exist along a continuum ranging from having each method be unique to a specific programming language to having methods which are general across all programming languages. Part of the hypothesis about the existence of methods is that, in fact, methods are specific to groups of programming languages which share common data and control structures. Thus, the same set of methods would be used for programs in FORTRAN and BASIC while a substantially different set would be used for programs in LISP.

Behaviorally, method-finding behavior should be identifiable by the occurence of the following types of behavior in a protocol of work on a programming task:

1. Statements of a general solution to the problem, in terms which are not constructs or legal statements of the programming language. Often, the statements are preceeded by phrases such as "the way I would do this would be ..."

2. Statements noting the similarity of this problem to one solved previously.

3. Solutions to the problem stated in a language other than the one in which the program will be finally written, if no attempt is made to check or verify the syntax of the lines of code. (This is meant to cover coding in metalanguages or informal, design languages; the reason for this particular distinction will be explained in the discussion of the coding process.)

The presence of an identifiable, associated behavior is by itself an insufficient basis for asserting that methods play a particular role in the creation of programs or that there is a distinct method-finding process. A somewhat stronger argument in favor of these hypotheses can be made from the following considerations: The space of possible programs is that of all possible statements or sequences of statements. Since this space is extremely large, constraints must exist which strongly limit the choice of statements to be selected for addition to a program. The constraints must be ones which operate across long segments of program, since statements written at one point in a program can determine the content of statements several hundred lines away. The problem statement cannot alone supply them since it does not sufficiently closely determine major characteristics of program structure, such as the choice of algorithm. Therefore, some source of information must exist which bridges between the problem statement and the writing of individual lines of code. The notion of method described here fills that role.

A Model of the Coding Process

At this point, the internal operation of the processes that make up the theory has only been presented in outline form. For the sufficiency of the theory to be evaluated, the structure of these processes must be more precisely specified. Resource constraints have so far permitted such a model to be constructed for only one of the processes, coding; details of this model are available in Brooks (Note 1); only the more salient aspects of this model will be presented here. As with the theory in which it is embedded, the goal in creation of the model is to provide a set of mechanisms which are sufficient to explain certain aspects of coding behavior. In particular, the model is intended to explicate two characterisics of coding behavior:

1. The order in which code is generated in a protocol, including the making of modifications and corrections.

2. The change, as the program writing progresses, of the programmer's knowledge about the program, as evidenced in the statements in the protocol.

The model is cast in the form of a computer program. "Input" to the model is considered to be a the method that the programmer has found for a section of code. As the model operates, the methods are converted into actual code, and the contents of certain data structures, which represent the programmer's knowledge about the program, are updated as this knowledge changes.

An Observational Basis for Constructing the Model

As is consistant with the general bias of this work towards describing and explicating individual behavior, the aim in constructing this model was to produce a structure that could explain individual episodes of coding behavior. Rather than producing such a structure by first generating an abstract model and then modifying it to fit individual episodes, an alternative strategy was followed. In it, the abstractions were made from a model that was initially constructed to fit an actual body of observed coding behavior. A description of these observations is a useful starting point for explication of the model.

The observations consisted of a set of 23 protocols by a single subject. The problems for which the protocols were collected were a set of 23 short problems, all of which involved manipulations on an array of 1ØØ numbers, called L. A second array, called M, could be used to indicate information about which manipulations had been performed. An example of one of the problems is:

> Find all the odd numbers in L and move them to the beginning of the array. Set the corresponding positions in M to 1.

While working on these problems, the programmer could use both paper and pencil and a 1Ø character-per-second, hard-copy computer terminal connected to an interactive computing system that the subject had used frequently before participating in the study. Alternation between the two could be made as frequently as the subject desired. His behavior while working was recorded using a throat microphone and a video tape recorder with the camera placed behind the subject.

For each program, the programmer was given a printed description of the problem to be programmed and the name of a file (dataset) on the computing system which contained code to read in the values of L and to zero M. He was instructed to write the program in the FORTRAN language and then to debug and run it. While performing these tasks, he was asked to "talk aloud" about what he was doing.

The subject employed in this study was a graduate student in computer science with more than 10 years of programming experience. This included writing several interpreters and assemblers as well as extensive experience in more than 10 programming languages. Additionally, he had three years experience teaching introductory programming courses.

Transcriptions of the tape recordings together with the subject's notes and the print-out from the terminal provided the basic data on the subject's behavior. The audio content of the tapes was transcribed and annotated, using the video information, to indicate the correspondence between writing or typing and the spoken material. A brief example of one of these protocols is:

S35: Then we'll say switch it with the next odd.

S36 Put a zero here.

A37 Changes NEXTODD=1 to NEXTODD=0.

S38 And we'll say K equals L sub I.

A39 Writes K=L(I)

Lines beginning with "A" indicate annotation of an action.

The breaking of spoken information into lines in the transcription was made according to two rules: First, a break was made whenever the subject paused in speaking, even if it was in the middle of a word or phrase. Second, if speech was relatively continuous, breaks were made between major clauses. The segmentation into lines in the protocol was, thus, intended to give a rough indication of the conceptual units used by the subject.

Since these 23 protocols were used to construct the model, a more precise characterization of them in comparision with other programming behavior is useful background for the model. One dimension along which such a characterization can be made is time, and timings were obtained using the counter on the video tape recorder. (Absolute accuracy varied depending on the amount of tape on the reel, but relative accuracy was .3%, or an absolute error of 5 seconds in a 25 minute protocol.) These figures are summarized in Table 1:

	Mean	Standard Dev.	Range
Writing time.	25.9	18.Ø	3.8-66.8
Debugging time. (minutes)	15.Ø	18.1	Ø.8-78.9
Ratio of times.	3.13	2.81	Ø.7-12.7
Lines of code written.	24.4	16.96	9-74

Correlations

Lines written x writing time	r=.69	p=.ØØØ5

Lines written x debugging time r=.75 p=.ØØØ5

Table 1

Writing time was defined to end and debugging time to begin when the subject first attempted to compile the program. The ratio of times refers to the ratio of writing time to debugging time for each problem.

In viewing these statistics, two items are noteworthy. The first is that even though the problems all involve only manipulations on one array, they differ widely in difficulty, with a 1 to 26 ratio between total time for the easiest and hardest problems. This finding is comparable to that found in the Grant and Sackman (1967) study.

Second, in these programs, writing time exceeds debugging time by as high as a 12 to 1 ratio. This is in extreme contrast to the results of other investigators (Youngs, 1974; Rubey, 1968) in which debugging time almost invariably exceeds writing time, occasionally by as much as 4 to 1. The source of this difference is hard to pinpoint, buti it may lie in the small size of the problems, the availability of "canned" code for input and output, or in the programmer´s claim to use the techniques of "structured" programmming cited earlier.

Further insight into the nature of this data may be obtained by classifying the statements in the protocol according to the scheme specified by the theory. The following tables gives the result of this classification:

Process	Occurence/Problem	Time spent (secs.)	% time
Understanding	1.39	1Ø1.4	6.9
Method-finding	1.19	313.4	21.2
Coding	1.87	1Ø6Ø.5	71.9

The Cccurence/Problem column indicates the mean number of times each process occurred per problem. The Time spent column indicates the mean total time spent on that process in each problem, while % Time is the percentage of total writing time accounted for per problem.

Note that both method-finding and coding occur only about twice per problem, and that coding accounts for more than two-thirds of the time spent. Using this information, a characterization of these problems is that they are easy to understand and that it is easy to find a solution method for them, but implementation of the solution is often quite difficult.

Architecture of the Model

The essential architecture of the model is adopted from the general theory of human problem solving of Newell and Simon (1972). Their theory specifies a short-term memory (STM) with a fixed, small capacity (less than 3Ø?) of chunks or symbols, and a long-term memory (LTM) of large or infinite capacity. Each symbol in STM "can designate an entire structure of arbitrary size and complexity in LTM " (Newell and Simon, 1972; p.795). Access times for STM are asserted to be quite short, on the order of a tenth of a second, while those for LTM are on the order of several seconds. (Readers are refered to the original reference for arguments in favor of these sizes and capacities.) Additionally, problem-solvers may make use of external memories (EMs) such as chalk boards, paper and pencil, etc. Capacities and characterizations of EMs will vary, but all will be dependent on the presence of appropriate access information in STM or LTM.

Problem-solving behavior in the theory is controlled by a production system. A producton system consists of a set of pairs of conditions and actions to be performed when the conditions are met. An appropriate resolution principle is employed to insure that only one set of actions is taken at a time. Executing the actions results in some change in the state of the data that are checked by the conditions, so that as the system operates, different sets of conditions are met and different actions are invoked.

In the Newell & Simon theory, the production system is considered to be part of LTM. The data for the conditions are the contents of STM. The theory further specifies that the production system is the only internal control mechanism for problem-solving behavior; an extensive defence of the psychological suitability of this structure for modeling human behavior is given in Newell & Simon (1972;p.804).

In the particular implementation of this theory used for this model of coding, the STM is presumed to consist of a fixed number of ordered slots. When new elements are introduced as the result of production actions, they are placed into the first slot, and each of the other elements is moved down one slot; the element that was previously in the last slot is lost off the end. In addition to adding new elements, the contents of STM may be altered by two other types of production actions: modification of elements already in STM, and moving elements into the first slot of STM that are already in some other slot. Brooks (Note 1) discusses the memory phenomena that are realizeable with such a structure, as well as the size requirements for the STM.

The production system used in the model is written in a formalism that is a variant of the PSG system (Newell & McDermott, 1975). In this variant, the invoking conditions are always tested in a fixed order, and the first production whose conditions are met is executed. The conditions are specifications of elements in the STM that are described in a pattern language. This pattern language permits the specification of elements in terms of their general form, as well as allowing the conjunction, disjunction, and absence of elements to serve as part of the invoking conditions.

An example of the kind of rule that can be written in this language is:

Conditions:

1. PLAN-ELEMENT (IF *REST*)
2. OLD-PLAN-ELEMENT (LOOP-THROUGH *ANY*)

Actions:

1. Remove the item matching condition #2.
2. Add GOAL TEST *REST* To STM.
3. Rehearse the item matching condition #1.

The *REST* and *ANY* tokens are patterns which match, respectively, the rest of an item and any single token in an item. Thus, the first condition would match either PLAN-ELEMENT (IF ODD-NUMBER) or PLAN-ELEMENT (IF GREATER SIZE, CURRENT SIZE).

As in the general theory, the production system is contained in LTM. In addition, LTM contains another structure called MEANINGS which is used to contain information about quantities and data structures used in a program for which code is being written. The reason for requiring a separate structure for this information is that the information of this kind is used over too long a period to reasonably assume that it is kept in STM. Assuming it to be stored in the production system would require the specification of mechanisms for allowing the production system to modify itself as information about the program being written changed. In the absence of such mechanisms, a separate LTM structure has been used in the model with expectation of eventually incorporating it into the production system. MEANINGS is assumed to have an associative structure in which a quantity or structure is accessed by giving enough of its attributes to identify it uniquely. Thus, a variable may be accessed by specifying that it is being used to keep track of elapsed time and that it is of type, real, without specifying that its name is ETIME. The particular set of attributes used will vary from subject to subject and problem to problem. Information is entered into MEANINGS and retrieved from it only by actions of the production system.

In addition to the STM and LTM memories, the model makes use of an "external memory" (EM) called CODE, which represents the information that the programmer has already written on paper or typed on a terminal. The basis for proposing such a structure is to represent the information that a programmer obtains from re-reading code that he has already written. Again, as in the case of MEANINGS, access to CODE is only via actions of the production system which adds new code to it and retrieves code that has already been written by copying information about it back into STM. Note that this means that information can only be retrieved from CODE if the contents of STM causes the necessary production rules to be used. In one sense, CODE contains the "output" of the model, since it is the receptacle for the code that the model generates.

Figure 1 indicates the overall structure of the model

- insert Figure 1 here -

Operation of the Model

As was discussed earlier, the action performed by the model is to take parts of a method and to convert them, via the symbolic execution process, into actual programming language statements. The production system, as center of control for the model, is responsible for carrying out the symbolic execution cycle. In particular, it must contain rules for selecting, on the basis of the method and the effects of previous code, the next piece of code to write and, once this code is written, it must assign an effect or result to the piece of code. The specific behavior of the model is, thus, entirely determined by the particular rules that make up the production system.

In generating the model, production rules were written for a set of 4 segments of coding protocol, each selected from a different problem in the set of 23 problems described previously. (The criteria used for selection are described in Brooks - Note 1). The number of lines of protocol in each segment and the number of lines of FORTRAN that were written by the subject in each segment are given in the first two columns of Table 2.

-insert Table 2 about here-

For each of these segments, the model was presumed to be initialized by placing the method, expressed in a list-structured notation, into the STM. Figure 2 shows one of these methods expressed both in natural language and in system notation form:

- insert Figure 2 about here -

A set of rules was constructed which, when applied according to the production system convention, progressively add new code to CODE and update the contents of STM and MEANINGS as specified by the symbolic execution cycles. The set contained a total of 73 production rules for all 4 segments.

Before discussing the nature of these rules, it is worth enquiring how well these rules did their job. Of the 46 lines of code in the written in the 4 segments, the model generated all of them correctly, but 2 lines in one problem were generated in a different order by the subject than by the model. Since the rules were constructed from the protocols, it is not surprising that they do a good job both of generating the code in the same order as did the subject. More interesting, however, are the figures shown in the third column of Table 2, which indicate the number of rule applications necessary to generate code for each segment, and the fourth column, which indicates the ratio between lines of protocol and the number of rules. If the number of lines of protocol are considered to be a rough measure of the amount of cognitive processing done by the subject, then the ratios indicate how well this corresponds to the amount of processing done by the model. Only in one problem does the ratio differ appreciably from 2 to 1, and this was the only problem in which there was any difficulty in generating the code. Since no explicit attempt was made to obtain this correspondence while constructing the rules, it serves as an independent indication of rule accuracy.

The 73 productions or rules used by the model can be divided into 4 groups on the basis of function. The first group consists of rules for overall control and goal management. An example of the 8 rules in this group is NEXT-PLAN-ELEMENT-1 which is responsible for getting the next piece of a plan once the previous piece has been completed. It states:

"If there is no PLAN-ELEMENT currently in STM, but there is an OLD-PLAN-ELEMENT, then add the next element of the plan to STM as a new PLAN-ELEMENT."

The second group consists of rules for updating the contents of the CODE EM. These rules all take some segment of code that other productions have generated and use it to add to or replace previously written code. An example of this group, ADD-ON-CODE, can be stated as follows:

"If there is any code in the STM to be ADDED-ON and there is a pointer to the CODE EM and there is an indicator of the last code added then add the code to CODE, mark the code as WRITTEN-CODE in the STM, and rehearse the pointer to the CODE EM."

The third group consists of the 57 rules which are actually responsible for generating code by the symbolic execution process. As such, they are the most substantive part of the model. Two examples of these rules are METHOD-ORDER-1 and GOAL-LOOP-THROUGH-1. The first of these rules can be stated as:

"If part of the method is to order a list, then begin by setting up a goal to loop through the list."

The second can be stated as:

"If the GOAL to loop through a list, then make this the current goal, prepare code for a DO loop running the length of the list, which is retrieved from MEANINGS, and update MEANINGS with information about the loop, and the index and the labels used in creating the DO statement.

Place the effect of the code into STM."

In operation of the model, these rules would be applied successively. After the second rule had been used, another rule would be applied which would compare the effect of this loop with the method step. On the basis of the difference, other rules would be invoked which would complete the code for the sorting operation.

Overview of the Rules

An interesting perspective on the rules in this class can be obtained by looking at the comparative patterns of use of the productions in all 3 classes. All the production rules in the first two classes are used repeatedly in all 4 program segments; in fact; 5 of these rules account for just over half (53.5%) of total rules utilization across all 4 segments. In the third group, on the other hand, only 9 of the rules are used in more than one segment. In terms of the structure of the model, this is, of course, an expected state of affairs. In each segment, different code is being created and different behavior is exhibited by the subject, even though the same basic control cycle and structures are being used. The model reflects this by having the rules that actually create code be different for each segment.

In terms of evaluating the sufficiency of this approach, the total number of rules that would be required to model all 23 problems is of interest. Using the methods from 4 additional protocols, it was estimated that somewhere between 4 and 1Ø new rules will have to be added to the third group for each new production to be modeled, assuming that no new productions are added in the first groups. On this basis, somewhere between 11Ø and 23Ø productions will be needed in the third group. Since the problems done here tap only a small fraction of a programmer´s knowledge, the total number of rules necessary to represent all of a programmer´s knowledge must be on the order of of tens or hundreds of thousands.

Is this magnitude reasonable? An argument can be advanced that it is. First, becoming a good programmer usually takes, in addition to formal training, several years of practical experience in actually writing programs. The learning that takes place at this time can be viewed either as the acquisition of a large body of diverse knowledge or as learning to apply a limited body of knowledge in a diverse range of situations. Whichever view is taken, the length of time needed suggests that the total amount of information to be acquired is quite large.

Another problem solving task which seems to depend on the acquisition of a large body of information by the problem solver is chess (DeGroot, 1965; Chase and Simon, 1973; Simon and Gilmartin, 1973). Like programming, it takes several years to learn to do well and does not require special motor skills. Additionally, like programming, it does not seem amenable to explanation in terms of some small, closed set of mental primitives, such as the problem space concept (Newll & Simon, 1972) unless these primitives are defined at a very low, atomic level. Using a simulation based on memory and eye movement studies, Chase & Simon (1973) have estimated that a chess master can recognize some 31,000 basic or primitive piece configurations which can be combined to represent an entire position. If programming knowledge is viewed as being built up out of some sort of analagous primitives or components, then the number of them required should be of similar magnitude. If each rule is considered to represent one such primitive or component, then an estimate of a need for tens or hundreds of thousands of such rules is a reasonable one.

Sufficiency of the Model

At the beginning of the article, several important findings from research on programming were cited. The remaining task is to demonstrate the sufficiency of the model for explaining them.

One of the more striking findings in the early work on programming was the large inter-subject and inter-problem difference. On naive grounds, one would certainly expect large differences between experienced and inexperienced subjects, but not a large difference within subjects of the same general background, intelligence levels, and experience. Further, a programmer who was good at writing one kind of program would be expected to do about as well at other, similar programs. That both these expectations do not hold ought to be one of the more important things that this theory of programming behavior be capable of doing.

Both kinds of differences are, can, in fact, be handled in the context of the theory and model presented here. If programming behavior is seen as determined by rules, then differences among individuals must lie in differences in the rules they have available. These differences must, in turn, result from differences in the training or experience of each subject. Since each rule encodes only a small piece of knowledge, slight differences in such training or experience may produce considerable differences in individual rules. As an example, consider two programmers with identical backgrounds except for which of two optional assignments they did in one training course. One did the assignment involving an odd-even test while the other did the one involving a table look-up. When faced with a problem that requires use of an odd-even test, one programmer will already know how to write the test while the other will not.

Not knowing how to write the test directly does not, of course, mean that he will be totally unable to do the problem. Instead, he will try and do it by an indirect route. Say, he may remember if a number is odd, dividing it by 2, dropping the fraction, and multiplying it by 2 will yield a result differing from the original number by 1. Using this fact, he may be able to apply several other rules to eventually construct the test. Doing it this way will, however, certainly take more time and will probably offer more opportunity for a mistake. This theory thus provides a mechanism for explanation of relatively large behavioral differences on the same problem across programmers with very similar background and experiences.

A slight variation on this same mechanism also can be used to understand how problems which appear very similar in structure may, nevertheless, differ greatly in difficulty. Consider the two problems, "find all the perfect squares in an array" and "find all the odd numbers in an array." The only difference in the programs to solve these problems lies in the test performed on array elements. If, however, a programmer is unaware how truncation on integer division can be used to test for oddness, he will find the second problem vastly more difficult than the first.

Viewing code generation as being accomplished by a large body of independent coding rules can also serve as a basis for theories of both debugging difficulties and of differences in types of errors across programming languages. The starting point for such theories is the assumption that many or most errors are caused by incorrect or inadequate code generation rules, rather than by external distractions or other random events. During the generation of the code, these faulty rules introduce errors or "bugs" into the program. For example, a rule may be insufficiently specific in its invoking conditions and cause a piece of code to get written in a circumstance for which it is not appropriate. If the effect which gets assigned to this code also does not make the needed distinction, the program will not perform properly.

This assumption leads to theories of debugging which are based on the premise that the same basic knowledge and procedures are used in both debugging code and in writing it. It suggests that debugging is essentially a process of generating code a second time and comparing it against the original for discrepancies. Such a viewpoint may prove a fruitful one. It leads, for example, to the explanation as to why debugging may occasionally prove exceedingly time consuming in terms of recreation of the original bug when the code is being generated again. It also can be used to explain the finding that have erroneous output available does not substantially reduce debugging time (Gould and

Drongowski,) on the basis that such information would play only a minor role in this recreate and compare process.

The assumption also leads to theories about the relationship between errors and programming language features. Since the rules encode a programmer's knowledge about the features of a programming language, the characteristics of these rules will determine under what circumstances an error will be made. While these rules will vary across individuals, rules for writing a given programming language construct may share certain common characteristics. For example, most programmer's rules for writing a GOTO statement in FORTRAN may require that the label, which is the GOTO destination, be paired with a specific, intended computation before it is used. Such common characteristics could also be common sources of error, perhaps because they require information which could be easily lost from STM. If this is the case, then particular programming language features would lead to high error rates across programmers. Explanations of this kind may be particularly powerful because they offer the possibility of evaluating potential new programming language features by constructing or hypothesizing the kinds of rules that are likely to govern their use.

The remaining behaviors to be brought within the coverage of the model are those observed in connection with the work on new programming methodologies. The beneficial effect of imposing a heirarchical cognitive organization is a fairly direct consequence of assuming an STM with a capacity of a fixed number of units. Since the heirarchical structuring would permit more information about the program to be packed into each unit, fewer, separate units would be needed, and the possibility of losing a unit, causing errors, would be reduced.

In addition to providing a mechanism by which this basic phenomenon can be produced, the model also can be used to derive some related behaviors. Among the more striking of these is a prediction of the conditions under which one of the heirarchical organization techniques is likely to succeed or fail. In "top-down" programming (Yourdon, 1975), actual construction of the program follows the cognitive organization. The programmer begins writing a program by first specifying the top level of the heirarchy and then successively specifying lower levels until the actual programming code is reached. In terms of the theory this corresponds to proceeding sequentially through understanding, method-finding, and code-writing. Under what conditions will the programmer be able to proceed in this direct sequential fashion, without backtracking? This will only be possible if the programmer always has at hand an appropriate set of rules at each step. This will usually occur only if the problem is of a type that is very familiar to the programmer and if the programming language is one with which the programmer has considerable experience. Hence, the model predicts that only under these conditions can top-down programming be used successfully.

Research Methodology

A concluding comment on this theory concerns the plan or course to be followed in doing research under it. As mentioned earlier, the theory is a theory of specific individual, rather than general behavior; and it focuses on building models of the behavior of single individuals, as was done here. The question remaining to be addressed concerns the kinds of generalizations or broader conclusions that can come out of building a set of such inividual models. If such a set is built for the performance of the same task by different individuals, a different set of rules will, necessarily, be used for each model. The sets, however, will show similarities across individuals. Some of these similarities will hold across all the individuals in the set, while others will hold only for subgroups. The course of work within this theory could be aimed at both identifying such similarities and at establishing their precursers in terms of the education and backgrounds of the individuals involved. The theory thus offers a potential for unified research into both individual aggregate behavior.

Footnotes

<1>. The name, "coding," has been chosen for this procedure because "code" is the generic term used by programmers to designate sequences of computer instructions written in a programming language, regardless of whether these instructions are part of a program, subroutine, function, coroutine,or some other unit.

Reference Notes

1. Brooks, R. A Model of Human Cognitive Behavior in Writing Code for Computer Programs. Unpublished doctoral dissertation. Psychology Dept., Carnegie-Mellon University. 1975.

2. Newell, A. Personal communication. The elements of the Newell theory that are used here are (1.) Development of methods 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. (3) Further generation of code based on recognition process driven by previously written code.

Bibliography

Anderson, J.R. & Bower, G. Human Associative Memory V.H. Winston & Sons. Washington, D.C. 1973.

Boehm, B.W. Software and Its Impact: A Quantitative Assessment. Datamation 1973. 19(5):48-59

Boies, S.F. & Gould, J.D. Syntactic errors in computer programming. Human Factors 1974 16:253-257

Bruner, J.S., Goodnow, J.J. & Austin, G.A. A Study of Thinking. New York. Wiley. 1956

Chase, W.G. & Simon, H.A. Perception in chess. Cognitive Psychology 1973 4 55-81

DeGroot, A.D. Thought and choice in chess. The Hague: Mouton, 1965.

Dijkstra, E.W. The humble programmer. Communications of the A.C.M. 1972. 15:859-866.

Gold, M.M. Time-sharing and batch-processing: an experimental comparision of their values in a problem-solving situation. Communications of the A.C.M. 12 249-259. 1969.

Gould, J.D. & Drongowski, P. An exporatory study of computer program debugging. Human Factors 1974, 16:258-276

Grant, E.E. & Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. I.E.E.E. Transactions on Human Factors in Electronics HFE-8. 1967.

Hayes, J.R. & Simon, H.A. Understanding written problem instructions. in L.W. Gregg (Ed.) Knowledge and Cognition Lawrence Erlbaum Associates. Ptomac, Maryland. 1974.

Hewitt, C. Description and theoretical analysis (Using Schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Unpublished doctoral dissertation. Department of Mathematics. M.I.T., 1972.

Knuth, D.E. An empirical study of FORTRAN programs. Software- Practice and Experience 1(1971),1Ø5-133.

Litecky, C.R. & Davis, G.B. A Study of errors, error-proneness, and error diagnosis in COBOL. Communications of the A.C.M.

Newell, A. & Simon, H.A. Human problem Solving. Prentice-Hall 1972. 1976. 19(1):33-37.

Rubey, R.J. A comparative evaluation of PL/1. Datamation December, 1968.

Simon, H.A. & Gilmartin, K. A simulation of memory for chess positions. Cognitive Psychology 1973, 5,29-46.

Schatzoff, H., Tsao, R. & Wiig,R. An experimental comparision of time-sharing and batch processing. Communications of the A.C.M. v.10(5) 1967.

Sime, M.E., Green, T.R.G. and Guest, D.J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-machine Studies 5(1) 105-113. 1973.

Smith, L.B. A comparision of batch processing and instant turnaround. Communications of the A.C.M. v.10(6) 1967.

Sussman, G. A computational model of skill acquisition. Unpublished doctoral dissertation. Massachusetts Institute of Technology. 1973.

Sussman, G.J. & McDermott, D.V. Why CONNIVING is better than PLANNING. (A.I. Memo No. 255A) Artificial Intelligence Laboratory. M.I.T. 1972.

Winograd, T. Understanding natural language. Cognitive Psychology 1972. 3, 1-191.

Youngs, E.A. Human errors in programming. International Journal of Man-machine Studies 1974. 6:361-376.

Yourdon, Edward _Techniques of Program Structure and Design_ Englewood Cliffs: Prentice-Hall 1975

1. Create a pointer to the next odd number, starting out at the
   beginning of the array.
2. Loop through the array.
3. Test each element to see if it´s odd. If it is,
    Increment the pointer to the next odd.
    Swap the element it points to with the element that was just
     found to be odd, which was pointed to by the loop index.
    Set the corresponding element in M to one.

```
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))
5. (SWAP-AND-INCREMENT
     ((ARRAY-ELEMENT (LIST OF NUMBERS)(VARIABLE
 (LOOP-INDEX)))
      (ARRAY-ELEMENT (LIST OF NUMBERS)(POINTER (NEXT ODD))))
      (POINTER (NEXT ODD))))
6. (SET (ARRAY-ELEMENT (AUXILLARY ARRAY) (POINTER (NEXT
 ODD)))
7. (END! (NOT-EVEN-PARITY))
8. (END! (LOOP-THROUGH))
```

Figure 2

The top portion of the figure shows a method expressed as English text.
The bottom portion shows the same method as it is represented in the model.

Problem	Lines	Cycles	Ratio	Code
#1	38	6Ø	1.6	9
#2	47	65	1.4	1Ø
#3	8Ø	169	2.1	14
#4	89	11Ø	1.2	13

Table 2

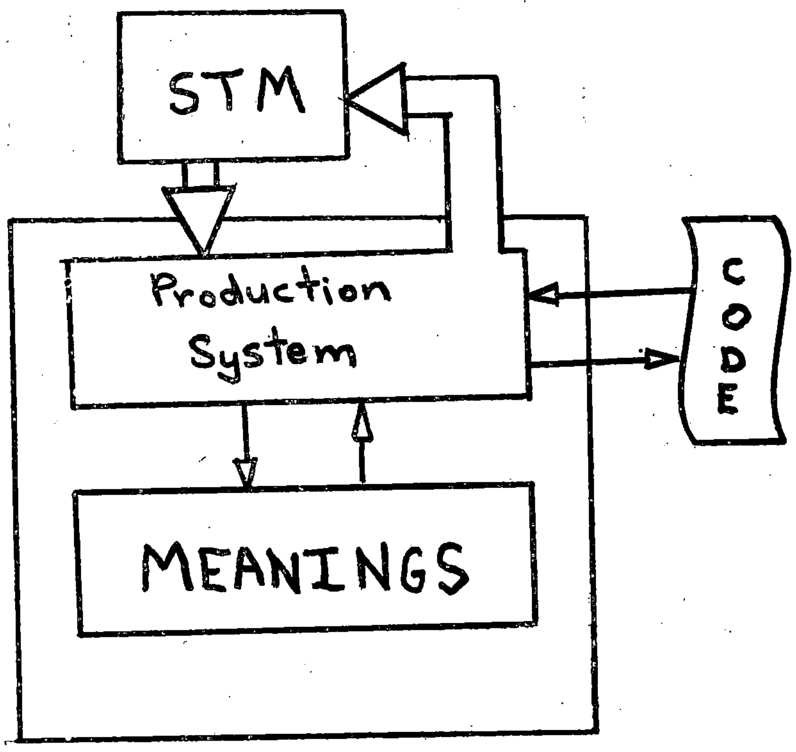

Figure 1.