Int. J. Man-Machine Studies (1983) **18**, 543–554

Towards a theory of the comprehension of computer programs

RUVEN BROOKS

International Telephone and Telegraph Advanced Technology Center, 1 Research Drive, Shelton, Connecticut 06484, U.S.A.

(*Received 5 May 1981, and in revised form 15 September 1982*)

A sufficiency theory is presented of the process by which a computer programmer attempts to comprehend a program. The theory is intended to explain four sources of variation in behavior on this task: the kind of computation the program performs, the intrinsic properties of the program text, such as language and documentation, the reason for which the documentation is needed, and differences among the individuals performing the task. The starting point for the theory is an analysis of the structure of the knowledge required when a program is comprehended which views the knowledge as being organized into distinct domains which bridge between the original problem and the final program. The program comprehension process is one of reconstructing knowledge about these domains and the relationship among them. This reconstruction process is theorized to be a top-down, hypothesis driven one in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation.

1. Introduction

Computer programming is a set of problem solving tasks in a semantically rich domain that are usually performed by professional problem solvers who acquired large amounts of domain-specific knowledge. In this regard, it resembles such other tasks as thermodynamics problem solving (Bhaskar & Simon, 1977), physics problem solving (Larkin, McDermott, Simon & Simon, 1980), chess (DeGroot, 1965), and medical diagnosis (Elstein, Shulman & Sprafka, 1978). The goal of research on these tasks is to explain how the domain-specific knowledge is organized and stored to be available for use in problem solving (Bhaskar & Simon, 1977), in contrast to studies of laboratory tasks such as cryptarithmetic (Newell & Simon, 1972) which aim to elucidate the fundamental problem solving processes themselves.

The focus of attention of this paper is one particular computer programming activity, the comprehension of a completed program. The reason for the choice of this activity is that it plays a role in nearly all software tasks. Its importance in maintenance activities is obvious, but it also plays an important role in many phases of initial software creation. For example, code reviews, debugging and some approaches to testing all require programmers to read and comprehend programs.

The theory presented here is intended to provide a set of mechanisms and relationships to explain the most salient aspects of program comprehension behavior. The behavioral differences to be explained include the following:

1. *The effects of differences of the task performed by the program on comprehension.* Why do programs that perform different computations vary in comprehensibility, even

0020-7373/83/060543+12$03.00/0

though intrinsic properties of the programs, themselves, such as length and complexity measures, are the same?

2. *The effects of differences in the program text.* Why, and under what conditions, do programs that are of different lengths languages, etc., differ in comprehensibility even though they perform the same computation?

3. *The effects of differences in the task being performed by the programmer.* Why might the comprehension process vary depending on whether the programmer was doing, say, modification as opposed to debugging?

4. *The effects of individual differences.* Why does one programmer find a program easier to comprehend than does another?

In presenting a set of mechanisms and relationships, the intent at this time is not to make specific predictions about these differences; rather, it is to show that the proposed mechanisms are sufficient to explain or model this variability. In this regard, this work takes a Theory Demonstration (Miller, 1978) approach.

2. A theory of how programs are comprehended

The major points of this theory can be summarized as follows.

1. The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain.
2. Comprehending a program involves reconstructing part or all of these mappings.
3. This reconstruction process is expectation driven by the creation, confirmation and refinement of hypotheses.

2.1. THE NATURE OF PROGRAM COMPREHENSION

The starting point for this theory is an analysis of the organization of knowledge used by a programmer when he or she writes program. In this view of the programming process, writing a program involves constructing mappings from a problem in some domain into the text of a program. As an example, consider a cargo-routing problem. In the application or problem domain, the objects are cargoes that have destinations which must be reached within time and cost constraints and there are means of transportation that carry these cargoes with various time and cost parameters. Before a program can be written to solve the problem, numbers must first be assigned to the cost and time elements, and identifiers (which also may be numbers) are assigned to cargoes and destinations. This results in a new knowledge domain, one in which the objects have become numbers.

In order to use the numbers in the program, an algorithm must be selected. This results in the creation of still another domain, one in which mathematical objects such as trees or matrices, are operated upon, and in which operators such as "invert a matrix" are used. Translation of the algorithm into a programming language creates yet another domain, with data structure implementations and the primitive operations of the programming language. Finally, execution of the program produces a domain in which the objects are the contents of memory locations and the operations are those of the hardware.

While this example has included five modeling domains, these same five would not necessarily be present in every task or for all parts of a task. If the task is to implement a particular algorithm for a general purpose routine, say a matrix inversion subroutine, there will be no domain corresponding to the original domain of cargoes and routes in the example. On the other hand, use of a primitive operation of searching a tree at one level might require use of an intermediate domain, containing methods of tree traversal, before the knowledge could be represented at the programming language level.

Using this idea of knowledge domain, the task of understanding a program for a programmer becomes one of constructing, or reconstructing, enough information about the modeling domains that the original programmer used to bridge between the problem and executing program. For example, if the programmer is given the task of modifying the dynamic programming algorithm in the cargo rate program the objects and operations in the domain of the original algorithm must first be learned about. Then, the programmer must find out about how these objects and operations appear in the particular programming language in which the program is written. If the algorithm is sensitive to issues of arithmetic precision, the programmer will also have to learn about the characteristics of the hardware on which the program will be run.

This construction or reconstruction involves the acquisition of two types of information.

Within each domain, there is information about the basic set of objects, including their properties and relationships, the set of operations that can be performed on these objects, and the order in which these operations take place.

Between domains, there will be information about the relationship between objects and operators in one domain and those in a nearby domain. These relationships will generally not be one to one, and new operations in one domain may be built up out of both objects or operands in a previous one. As an example, the operation of traversing a tree in an algorithm domain may map onto a sequence of statements (operations) and a set of variables (objects) in the programming language domain.

2.2. THE PROCESS OF DOMAIN RECONSTRUCTION

2.2.1. Hypotheses

One view of the process of understanding a program is that it is fundamentally bottom-up and that the programmer begins by looking at the individual lines of code or at groups of lines and assigning them interpretations. The interpretations are in terms of domains close to that of the program text, such as that of simple algorithms.

These interpretations are aggregated to provide higher interpretations for larger and larger segments of code until the entire program is understood (Shneiderman & Mayer, 1979; Basili & Mills, 1982).

While most programmers will recall instances when they have attempted to behave in this way, this theory asserts that this sort of inductive behavior is only a degenerate special case of a more powerful process. This process is based on the successive, top-down refinement of hypotheses about other knowledge domains and their relationship to the executing program.

These hypotheses are generated from the programmer's knowledge of the task domain and of other domains that might be related to it. The refinement process is

a progressive, iterative one which is based on the information extracted from the program text and other sources and which may involve the generation of subsidiary hypotheses. Using the transportation problem as an example, the programmer may use his or her knowledge of the problem domain to hypothesize that the program takes rate equations and distance tables as input and uses a dynamic programming algorithm to calculate the output. Since the distance tables are likely to be large, and since most dynamic programming algorithms are likely to need information on only a small subset of the possible links, the programmer makes the further, subsidiary hypothesis that only part of the distance tables are kept in core at any point in time and that some kind of indexing scheme is used to retrieve the distances between individual pairs of cities. On looking at the listing, the programmer finds that random access I/O is used to read into two arrays, DIST and CITY, and concludes that DIST is used to hold the distance to each of the cities specified in CITY. Using this assumption about DIST, the programmer then searches for other uses of DIST. From these uses of DIST, a hypothesis is made about the specific dynamic programming algorithm. This, in turn, will lead the programmer to create further subsidiary hypotheses or to look for certain features of the program text. This process will continue until the programmer has sufficiently reconstructed the knowledge domains needed for the particular task that is being performed.

The nature of hypotheses. As this example illustrates, hypotheses do not, in general, explicitly name the components—the objects, operators, and operation sequences—that make up the domain being reconstructed. Rather, they contain just descriptions of the components in terms of the functions they perform. In the example, the hypotheses did not name the specific algorithm being used, but, rather described it in terms of the class of algorithms to which it belonged, dynamic programming algorithms. Thus, they are very much like the schemas that Jeffries, Turner, Polson & Atwood, (1981) have proposed are used in designing software.

2.2.2. *Hypothesis generation and revision*

This theory asserts that the hypothesis generation and verification process begins with a primary hypothesis that is generated as soon as the programmer obtains some information about the task that the program performs. Though the level of detail in this primary hypothesis will vary across individuals and tasks, it will specify at least the global structure of the program in terms of inputs, outputs, major data structures and processing sequences. It is claimed that the primary hypothesis is often created by the time the programmer has heard only the name of the program or a brief descriptive phrase; only if the domain is utterly unfamiliar will primary hypothesis formation wait until the programmer has seen an extensive description of the program or until the program text itself has been seen.

The basis for this claim lies in the extreme power of hypotheses for reducing the space of program explanations. Even a simple phrase like "master file update" strongly suggests the structure of the program; in this case, it would lead the programmer to think about two input sources, one for the master file and one for update information. Using this guess, the programmer could then immediately direct attention while reading further description to deciding which input source performs which function. Thus, the earliest possible formation of hypotheses, no matter how weak the hypotheses are, may be of considerable aid in further understanding of the program. (The issue of hypothesis correctness will be dealt with below.)

The hypothesis generation process. In order to be most effective in reducing the space of potential program explanations, the hypotheses must suggest the most likely program structures and organizations for solving the problem at hand. One readily available source for hypotheses fulfilling this requirement is for the programmer to appeal to his or her own program writing processes and knowledge. In an earlier work (Brooks, 1977), a model was proposed of this program writing process: for purposes of the discussion here, however, the property of this model that is of importance is its stress on the role of learned patterns or structures in the program writing process. In this model, the writing of programs is seen as primarily a process of retrieving known structural units (of various degrees of granularity) and assembling them, perhaps with modification, into programs.

In this case of hypothesis generation, it is proposed that the same units are retrieved and used for hypothesis formation as for new code generation; however, in contrast with new program development, the assembly and modification phases are greatly reduced. In general, the assembly process is only carried out far enough to result in a hypothesis that is verifiable against the code. Thus, in the file update example, either new code generation or new hypothesis generation might result in retrieval of a unit specifying reading the updates file a record at a time and searching for a master file entry, but, for use as a hypothesis, no further refinement would be carried out until the programmer actually looked at the program code.

While an explicit model for the units selection process is not needed here, it is worthwhile to note the heavy dependence this process must have on the programmer's knowledge of the problem domain and of domains that map between it and the program. As an example, consider the effect knowledge about the problem domain might have in the file update situation. The most common assumption might be that the master file is ordered by identification number. If, however, it is known that the distribution of updates is very skewed so that, say, 20% of the master file entries receive 90% of the update activity, then the master file might best be ordered by frequency of access. Since this ordering method is rather unusual for most file ordering tasks, someone reading the program is likely to hypothesize it only if they are familiar with the peculiar distribution of updates in this domain. As this example illustrates, knowledge about the problem domain can be critical in making hypotheses about the program structure.

In terms of its effect on reducing the space of program interpretations, the primary hypothesis must balance being general enough to cover all likely programs while still constraining search by ruling out unlikely candidates. As an illustration, in the file update example, a useful hypothesis would be that there are at least two separate kinds of input and that each kind is read separately. A hypothesis that specified only that the program read some input and produced some output would be too vague to be of any use while one that specified that the input was in two separate files would unnecessarily rule out a program in which the two kinds of input followed each other in the same file. It is theorized that the ability to generate hypotheses of appropriate specificity will be a function of both the overall skill level of the programmer and of experience with the particular kind of problem that the program is solving.

2.2.3. *Verification and the hypothesis hierarchy*

In order to verify or validate a hypothesis, the programmer must find evidence in the code or the associated documentation that supports the hypothesis. Since the primary

hypothesis is usually, and appropriately, global and non-specific, the programmer will almost always find it impossible to verify his hypothesis directly against the program code. This theory specifies that what takes place, as illustrated earlier, is the construction of subsidiary hypotheses having a hierarchical structure with those lower in the hierarchy being more specific and less functional, and with lower elements in the hierarchy adding specific detail and concreteness to the elements above them.

These subsidiary hypotheses are usually created in a top-down, depth-first manner (to minimize memory loads?), so that a part or function of a hypothesis at one level is further refined before the rest of the hypothesis at that level. At the beginning this creation of the subsidiary hypothesis is done using only the programmer's knowledge of the domain and of similar programs. Eventually, the lowest point in the hierarchy may be refined enough to be verified against the program text or the documentation and the verification process may yield information about the program which can be used elsewhere for subsidiary hypothesis generation.

(*a*) *The basis for hypothesis verification.* The condition that causes the process of creating further refined subsidiary hypotheses to terminate is that the level of detail of the hypotheses is sufficiently close to that of the program text of documentation so that a comparison can be made. This occurs when the operations or data structures specified in the hypotheses are ones which the programmer can associate with features or details visible in the program or documentation which are typical indicators of the use of the particular operation or structure. For example, a typical indicator for the use of a sort of array elements would be a section of code in which the values of array elements were interchanged, particularly if this code were located inside a pair of loops

```
      DO 10 I = 1, NN
      DO 10 J = 1, MM
        :
      TEMP = A(1)
      A(I) = A(J)
      A(J) = TEMP
        :
10    CONTINUE
```

Sets of features that typically indicate the occurrence of certain structures or operations within the code will be referred to as "beacons" for that structure or operation. Obviously, there can be multiple beacons for a given structure or operation, and different beacons may indicate the presence of these structures with different probability. Further, the same feature may participate in beacons for a variety of structures or operations.

(*b*) *The verification process.* The actual process of verification involves scanning or searching the program text or the associated documentation for beacons that confirm and elaborate the hypotheses under consideration. The particular strategy used in making this search varies across individuals and will be discussed in a subsequent section. Regardless of strategy, however, it theorized that information collection during the search will be broad, rather than focused only on the hypothesis at hand. Thus, strong beacons for any structure will be noticed, whether the structures are currently hypothesized or not.

The information collected in this way can later play two roles in the hypothesis verification process. The first is that it can reduce the need for searching the program text to verify successive hypotheses. If the programmer remembers the occurrence of a beacon for a structure or operation different from the one being sought, there will be no need to search for it if the structure is later hypothesized.

The second role that can be played by information collected during passes through the code is to suggest new subsidiary hypotheses. If certain structures are known to exist in the code, then subsidiary hypotheses can be made which link these structures to more global hypotheses. Some of these new hypotheses will be created and satisfied at the same time; for example, if the programmer saw beacons distinctive of a binary search and thought that there was, indeed, a search in the program and that it was a binary search.

(*c*) *Binding.* Up to this point, the effect of the discovery of a strong beacon for a hypothesized structure or operation has been described only as acting to confirm the hypothesis. This theory proposes, however, that the discovery of a beacon results in a further refinement and specification of the hypothesis itself. When a hypothesis is first generated it consists of a set of functional specification of operations and structures; the discovery of a beacon permits indicators or features from the code to be bound to the functional specifications. For example, if the subsidiary hypothesis was that the program must contain a sort, then the information that would be bound to this hypothesis after discovering the appropriate beacon in the code would be that the sort was performed in certain lines of the program, that it was a bubble sort, and that the name of the array sorted was LDAT.

When hypothesis fails. The general structure for the program proposed by this theory for the program understanding process is that the programmer first forms a primary hypothesis about the overall functions performed in the program. From this primary hypothesis, a cascade of subsidiary hypotheses are formed until a level is reached at which the programmer can verify the code against beacons in the program text. When the beacons for a particular function are found, information about that section of the program is bound to the hypothesis. In the course of searching for a particular set of beacons, the programmer may uncover other beacons which lead to the generation and verification of other, subsidiary hypotheses. When this process is complete, the end product will be a hierarchical structure with the primary hypothesis at the top, subsidiary hypotheses below, and each segment of program bound to a subsidiary hypothesis, with no unbound parts of the program. If this state is reached, the programmer is considered to completely understand the program, at least in regard to attributing a function to each statement at the source language level.

In a few rare instances the programmer will select a correct primary hypothesis, derive subsidiary hypotheses from it, and be able to bind code uniquely to each subsidiary hypothesis without any errors. In most programs, however, problems will occur which will manifest themselves as one of three symptoms: the programmer will be unable to find code to bind to a particular subsidiary hypothesis, or the same code will be bound to two different hypotheses, or there will be some parts of the code which cannot be bound to any hypothesis.

An example of the first symptom would be if the programmer has hypothesized that the program contains a sort, but is unable to find beacons for known and familiar

sorts. This may occur either because the hypothesis is wrong, and the program does not contain a sort, or because some very unfamiliar type of sort is being used. An example of the second kind of error might be the conclusion that the same array is being used to hold both ordered master file records and unordered update records. This occurs because the features of use of that array could satisfy either hypothesis. Finally, an example of the third type of error would be the presence of unexplained lines of code inside what the programmer believes to be a simple insertion sort, a situation which might occur because what is supposed to be a sort is, in fact, something else, or because more than one function is actually being performed by that section of code.

All of these kinds of errors could be cured either by adopting different hypotheses or by altering and adding to the bindings of code to hypotheses. The theory must, therefore, incorporate some mechanism for representing the process by which the programmer decides either to change his ideas about what the program does or to look more closely at the program to see if it does, indeed, do what he thinks it does. A structure which seems to have the necessary properties is to assume, as was mentioned earlier, that each binding has some strength associated with it, depending on the number and specificity of beacons used to make it. These bindings are backed up the hierarchy tree in much the same way as position values are backed up a game tree. In deciding whether to alter a hypothesis, the programmer first attempts to generate an alternative hypothesis consistent both with the code and with the hypothesis above it in the hierarchy. If no such hypothesis is available, as may often be the case, then an assessment is made of the other hypotheses hanging off the parent of the current hypothesis. If their bindings are relatively strong, then the programmer will attempt to process the code further, in the hopes of finding an interpretation of it which is consistent with the hypothesis. This further processing will involve closer and more detailed inspection of the code, for example, checking the code against the comments. On occasion, it may result in the same kind of line-by-line inspection described earlier.

If the hypotheses hanging below a given parent receive mixed or weak support in the code, then the programmer will conclude that it is the parent hypothesis which is at fault. If a suitable alternative can be constructed, an attempt may then be made to verify the alternative. This kind of backing up may continue to the level of the primary hypothesis, but only if suitable alternatives and their subsidiaries are available. If they are not, the programmer will continue efforts towards verification of the hypotheses at hand until success is obtained or until some effort limitation is exceeded.

Since most programmers are unable to generate a very large number of alternative ways of performing the same computation, the net result will be that programmers will rarely, if ever, abandon a hypothesis and backtrack. Instead, a more common sign of difficulty in hypothesis verification should be repeated attempts to interpret the code.

3. Implications

3.1. WHY DO PROGRAMS DIFFER IN COMPREHENSIBILITY?

3.1.1. The role of the problem

Considerable evidence already exists that the control constructs used to express a program can have an effect on its comprehensibility (Green, 1977; Soloway, Ehrlich, Bonar & Greenspan, 1982). Work on software metrics has shown that factors such as the number of identifiers, the number of statements, and the amount of branching also have an impact on program comprehension (Curtis, Sheppard, Milliman, Borst & Love, 1979).

What this theory suggests is that an even more powerful source of variation may lie in the tasks the programs perform. If comprehension of a program does indeed involve reconstruction of the relationship between the original problem and the program text, then the difficulty of comprehending a program will depend in part on the difficulty of comprehending the original problem. Moreover, the difficulty of the problem may bear no relation to the complexity of the resulting program; compare, for example, the complexity of the theory of one-way ANOVA with the simplicity of the programs to compute it. At least some experimental evidence for this position may be seen in the finding of Curtis *et al.* (1979) of significant differences in comprehensibility among programs with the same software metric values.

3.1.2. The role of documentation

(*a*) *Different kinds.* The text of a program and the associated documentation can be viewed as a collection of indicators to be assembled into beacons. Table 1 shows some of the more commonly used sources of indicators.

TABLE 1
Indicators for the meaning of a program

Internal to the program text
1. Prologue comments, including data and variable dictionaries
2. Variable, structure, procedure and label names
3. Declarations or data divisions
4. Interline comments
5. Indentation or pretty-printing
6. Subroutine or module structure
7. I/O formats, header, and device or channel assignments
External
1. Users' manuals
2. Program logic manuals
3. Flowcharts
4. Cross-reference listing
5. Published descriptions of algorithms or techniques

The interpretation and saliency of a particular indicator will, as described previously, be a function of the beacons it participates in and the saliency of these beacons for the particular hypotheses under consideration. Thus, it will, in general, be necessary to know the current hypothesis structure in order to predict exactly how these sources of information will be used in any specific situation. Nevertheless, several broad generalizations can be made about the use of these materials which will be statistically true across the range of program understanding situations. The first of these is that, since at the initial stages of hypothesis verification, the hypotheses tend to be more global and less detailed, the kinds of beacons and indicators needed will also tend to

be those related to the global structure of the program. The primary source of such global information is most likely to be materials such as design descriptions and user's manuals. As hypotheses become more detailed, lower level materials, such as variable names and program statements, are more likely to be the sources of important indicators.

(*b*) *Documentation may be programming language specific.* Another prediction concerns the interaction of documentational devices such as flowcharts or program design languages with the programming language syntax and semantics. Since programming languages differ in their availability of constructs to express different aspects of a computation, the kinds of documentation they require will differ. For example, LISP has built-in primitives for manipulating singly linked lists; PASCAL requires that the user build up these same operations out of primitive pointer operations. Documentation indicating the relationship between a sequence of PASCAL statements and the list operation they implement is likely to be useful for a PASCAL program. Conversely, LISP programs will require documentation for record structures which would be unnecessary in PASCAL.

(*c*) *More documentation is not necessarily better.* Redundancy in indicators occurs if multiple indicators contain essentially the same information. The impact this has on program comprehension may be mixed. On one hand, if the indicators are all of different types, the redundancy increases the likelihood that a particular indicator will be interpreted and used. On the other hand, if disproportionately many indicators redundantly present the same information, then they may act to obscure or overwhelm other indicators that contain unique information. Thus, it may be possible to over document or over comment a program.

It may also happen that one indicator contradicts another. In some cases, this is because there is an error in the current hypothesis which results in a misinterpretation of the indicator. For example, if an array is believed to be holding the key numbers of cities on a shipping route when, in fact, it is holding distances between cities, then a check to be sure the value of one array element is less than another will be seen as inconsistent. In other cases, the indicators will, in fact, be contradictory, such as when a comment disagrees with the code. Whichever the cause, the action taken by the programmer will depend primarily on the overall support for the hypothesis element being contradicted and, only secondarily, on the particular type of indicator. Thus, if a piece of code contradicts a comment that is otherwise well supported, the programmer may choose to believe the comment and seek an alternative more consistent interpretation of the code.

3.2. HOW DO TASK DIFFERENCES AFFECT COMPREHENSION?

In general, comprehension of a program should follow the process described here independent of the task for which comprehension is required. Regardless of the task, it may be the case that the programmer must still completely reconstruct the original domains and their mappings. In the shipping problem, for example, what appears to be a bug in calculating shipping costs in the program may, in fact, be due to different methods for calculating air and sea costs in the original problem domain.

In general, though, different tasks will require varying levels of comprehension of different aspects of the program. Thus, a programmer whose task is to modify the output format will be more concerned with the output statements and less concerned with the major control structure than one who is attempting to find a bug that is causing the program to produce wrong values. When given a new program listing to look at, these task goals will affect the way the programmer searches the listing. For example, the programmer charged with modifying the output is likely to begin by searching for all the output statements in the listing. Manipulating the task goal should therefore produce changes in what the programmer looks at when first given a listing.

3.3. WHY DO PROGRAMMERS DIFFER IN THEIR ABILITY TO COMPREHEND PROGRAMS?

This theory predicts at least three distinct sources of differences in the ability of programmers to comprehend programs:

programming knowledge;
domain knowledge;
comprehension strategies.

Clearly, a programmer's ability to confirm hypotheses against code and to make appropriate refinements to hypotheses will depend on the programmer's knowledge of typical programming idioms and algorithms and the ways in which they are combined. A critical issue appears to be not only the total quantity of the knowledge, but also the ability to apply it in the appropriate point in the hypothesis verification process. If subsidiary hypotheses are unnecessarily restrictive, then the programmer may be unable to match them to what the code actually contains, even if the programmer might recognize the same code under a different hypothesis. This viewpoint is substantiated by the work of Atwood & Ramsey (1978) and Atwood, Turner & Ramsey (1979), and that of McKeithen, Reitman, Rueter & Hirtle (1981), which has shown that experts differ from novices more in their ability to organize information about a program than they do in the absolute quantity of information per se.

3.3.1. Differences in problem domain knowledge

While the contribution of differences in programming knowledge to comprehension has already received some study, this theory suggests that differences in knowledge about the problem domain, a factor that has not yet received much study, may account for an equally substantial share of the interprogrammer variance. Since domain knowledge is critical to the formation of the top level hypotheses, a programmer will have extreme difficulty in forming useful ones if he or she does not understand the problem the program is solving. Consider how much more difficult the comprehension process described by Basili & Mills (1982) would have been, had they not understood what an inverse quadratic interpolation was or the mathematics of computing one.

If this conjecture is correct, then it may be as important to document adequately the rationale behind the specifications for a program as it is to document the resulting program. Note that this is not the same as merely giving the requirements for the program; it must also include a history of the particular decisions that lead to these requirements, since this history may contain important clues for the development of correct hypotheses.

3.3.2. Differences in strategy

A final source of individual variation in program comprehension may be the strategies that the programmer uses to locate information in the program text. Even given the

same essential hypotheses, one programmer may attempt to validate it by tracing the subroutine calling hierarchy while another tries to locate input and output functions. While these differences are not likely to be as large as those from programming knowledge and domain knowledge, strategy differences may play a role in accounting for pathological cases of programmers who are exceptionally successful or unsuccessful at program comprehension.

References

ATWOOD, M. E. & RAMSEY, H. R. (1978). Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging. *U.S. Army Research Institute for the Behavioral and Social Sciences, Technical Report 78-A21.*

ATWOOD, M. E., TURNER, A. A. & RAMSEY, H. R. (1979). An exploratory study of the cognitive structures underlying the comprehension of software design problems. *U.S. Army Research Institute for the Behavioral and Social Sciences, T.R. 392.*

BASILI, V. R. & MILLS, H. D. (1982). Understanding and documenting programs. *IEEE Transactions on Software Engineering*, **SE-8** (3), 270–283.

BHASKAR, R. & SIMON, H. A. (1977). Problem solving in semantically rich domains: An example from engineering thermodynamics. *Cognitive Science*, **1** (2), 193–215.

BROOKS, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man–Machine Studies*, **9** (6), 737–742.

CURTIS, B., SHEPPARD, S. B., MILLIMAN, P., BORST, M. A. & LOVE, T. (1979). Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, **5**, 96–104.

DEGROOT, A. D. (1965). *Thought and Choice in Chess.* The Hague: Mouton.

ELSTEIN, A. S., SHULMAN, L. S. & SPRAFKA, S. A. (1978). *Medical Problem Solving.* Cambridge, Massachusetts: Harvard University Press.

GREEN, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, **50**, 93–109.

JEFFRIES, R., TURNER, A. A., POLSON, P. G. & ATWOOD, M. E. (1981). The processes involved in designing software. In ANDERSON, J., Ed., *Cognition and Problem Solving.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

LARKIN, J., MCDERMOTT, J., SIMON, D. P. & SIMON, H. A. (1980). Expert and novice performance in solving physics problems. *Science*, **208**, 1335–1342 (June).

MCKEITHEN, K. B., REITMAN, J. S., RUETER, H. H. & HIRTLE, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**, 307–325.

MILLER, L. (1978). Has artificial intelligence contributed to an understanding of the human mind? A critique of arguments for and against. *Cognitive Science*, **2** (2), 111–128.

NEWELL, A. & SIMON, H. A. (1972). *Human Problem Solving.* New York: Prentice-Hall.

SHNEIDERMAN, B. & MAYER, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer and Information Sciences*, **7**, 219–239.

SOLOWAY, E., EHRLICH, K., BONAR, J. & GREENSPAN, J. (1982). What do novices know about programming? In SHNEIDERMAN, B. & BADRE, A., Eds, *Directions in Human-Computer Interactions.* Hillsdale, New Jersey: Ablex, Inc.

Int. J. Man–Machine Studies (1983) **18**, 555–565

A note on the functional estimation of values of hidden variables—an extended module for expert systems

NICHOLAS V. FINDLER AND RON LO

Group for Computer Studies of Strategies, Computer Science Department, Arizona State University, Tempe, Arizona 85287, U.S.A.

(*Received 20 June 1982, and in revised form 20 September 1982*)

This article describes an extension of our work on the Generalized Production Rules System. In its original form, it could estimate at a given point of time or space the value of *hidden variables*—variables that can be measured only intermittently or periodically. In contrast, *open variables* are readily measurable any time. The system establishes stochastic, causal relations, *generalized production rules*, between known values of hidden variables and certain mathematical properties of the open variables' behavior. These rules are then used to make the point estimates.

We have now provided the system with the additional ability to estimate the *functional behavior* of the hidden variables. The system can serve as a domain-independent module to a knowledge-based expert system in need of such numerical estimates.

Introduction and brief summary of prior work

We have previously reported on various theoretical and practical aspects of the Generalized Production Rules System (GPRS) (Findler, 1980, 1982*a*, *b*). The reader is referred to the companion paper of this one (Findler, Brown, Lo & You, 1983), in which a detailed account is given of the background, motivation, methodology and the results of the project. However, a short summary of the underlying ideas of GRPS below will make this article understandable on its own.

Strategic decision-making aims at achieving one's own goals and preventing the achievement of the adversaries' goals over a sequence of confrontations. In order to do so, the decision-maker needs to know the values of relevant variables at various times. Some of these variables, the *open variables*, are readily measurable at any time. Others, the *hidden variables*, can be measured only at certain times, intermittently or periodically.

The rules in the knowledge base of GPRS connect causally- and stochastically-related open and hidden variables. Both the causes and effects can be open or hidden variables. The objective of the system is to provide increasingly sharp estimates of the values of the hidden variables as both the number and the quality of the rules increase.

The prediction or estimation of the hidden variable (HV) value is based on a subset of the mathematical properties of the open variable (OV) distribution. Let us assume that we have a sequence of values of an OV over time or space (called "lag variable" for reasons shown below). A part of the system, the morph-fitting program (Findler & Morgado, 1982), constructs a unique mathematical description of the behavior of

0020-7373/83/060555+11$03.00/0