Information and Software Technology 1995 **37** (7) 351–362

A conceptual model of cognitive complexity of elements of the programming process

S N Cant, D R Jeffery and B Henderson-Sellers

School of Information Systems, University of New South Wales, P.O. Box 1, Kensington, New South Wales 2033

A new approach to complexity metrics is described based not on empirical analysis of the final product, viz. the code, but on an understanding of the cognitive processes of the analyst or programmer as they approach and undertake the challenges of program development, modification and debugging. The resulting metric, the Cognitive Complexity Model, involves quantification of a number of cognitive processes, focused on descriptions of comprehension resulting from the twin processes of 'chunking' and 'tracing' used by software developers in an attempt to reach a cognition of a software system at the code level. A conceptual framework is given as well as some illustrative indicators of likely component measures together with areas needing further research.

Keywords: complexity, measurement, software, chunking, tracing, cognitive complexity

Software complexity research began in the 1970s around the same time as the push for structured programming was beginning. Both were the result of an increasingly large number of poor experiences among practitioners with large systems and, in consequence, the need for quality control for software became very strong. This quality control took two forms. Firstly it involved the development of new standards for programming and, secondly, it required the development of measures to monitor the complexity of code being produced so as to provide benchmarks and to permit poor quality sections to be modified or rewritten. The best known of the early approaches to complexity measurement were those of McCabe[1] and Halstead[2]. Despite subsequent criticisms of these metrics (e.g. Coulter[3]; Shen *et al.*[4]; Evangelist[5]; Nejmeh[6]; Munson and Khoshgoftaar[7,8]; Shepperd[9]) they are probably still the best known and most widely used (see Bowman and Newman[10]; Gill and Kemerer[11]). However, it should be noted that Halstead's Software Science metrics have not survived this criticism, being regarded by Card and Glass[12] (p 27) as a metric that 'practitioners can safely ignore'.

At present there is no unified approach to software complexity research. Curtis[13] (pp 96–97) examining the field lamented that 'Rather than unite behind a common banner, we have attacked in all directions'. He made a number of observations:

- there was an over-abundance of metrics;
- metricians defined complexity only after developing their metric;
- more time was spent developing predictors of complexity, where in fact more needed to be spent on developing criteria, such as maintainability, understandability and operationalizing these criteria*;
- for every positive validation there was a negative one.

There have been some improvements in the past decade. The rate at which new complexity metrics are appearing has fallen. A few metrics, such as those of McCabe[1], Halstead[2] and Henry and Kafura[23], have been the subject of a large amount of empirical and theoretical research (Zuse and Bollman[24]; Henry and Salig[25]; MacDonnell[26]; Lakshmanan *et al.*[27]; Khoshgoftaar *et al.*[28]; Mata-Toledo and Gustafson[29]). However, their dominance does not appear to be based on any impressive empirical success, but rather on the lack of any validated 'rivals'. Still today there are several theoretical approaches to complexity, none of which have been successful enough to claim sole rights to the banner of software complexity research (Kearney *et al.*[30]; Weyuker[31]; Tian and Zelkowitz[32]; Cherniavsky and Smith[33]; Munson and Khoshgoftaar[34]).

*Some work in this area has occurred, the most notable being McCall *et al.*[14]; Boehm *et al.*[15]; Bowen *et al.*[16]; Kitchenham and Walker[17]; Fenton and Kaposi[18]; Fenton and Whitty[19]; Fenton[20]; Fenton and Melton[21]; and Fenton[22].

At present there is no accepted definition of software complexity. The Macquarie Dictionary defines the common meaning of complexity as 'the state or quality of being complex' i.e., 'composed of interconnected parts' or 'characterized by an involved combination of parts ... an **obsessing notion**'. This definition is intuitively appealing. It recognizes that complexity is a feature of an object being studied, but that the level of perceived complexity is dependent upon the person studying it. For example, a person unfamiliar with an object may find it far more difficult to understand than someone who has studied the object many times before. However, the definition has two deficiencies as a technical definition upon which to base research. The first is that it ignores the dependency of complexity on the task being performed. As Ross Ashby (quoted in Klir[35] p 326) noted:

> ...although all would agree that the brain is complex and a bicycle simple, one has also to remember that to a butcher the brain of a sheep is simple while a bicycle, if studied exhaustively ... may present a very great quantity of significant detail.

The second problem with the definition is that it does not allow complexity to be operationalized as a measurable variable. The factors described as determining complexity are very general and therefore lack the specificity required for measurement. It was probably with these problems in mind that Basili[36] formulated his definition of 'software complexity' as: 'A measure of resources expended by a system [human or other] while interacting with a piece of software to perform a given task.'

Although this definition addresses the two deficiencies in the dictionary definition, it may be too broad. As the dictionary definition indicates, complexity is usually regarded as arising out of some characteristic of the code, on which the task or function is being performed and therefore does not include characteristics of the programmer which affect resource use independently of the code. A suitable compromise may be the definition proposed by Curtis[13] (p 102). He claimed that software complexity is: 'A characteristic of the software interface which influences the resources another system will expend while interacting with the software.'

This definition is broad enough to include characteristics of the software which interact with characteristics of the system carrying out the task, in such a way as to increase the level of resource use. However, it is not so broad as to include arbitrary characteristics of the programmer which affect his or her ability generally. This distinction is illustrated in Figure 1. The left and centre areas cover those

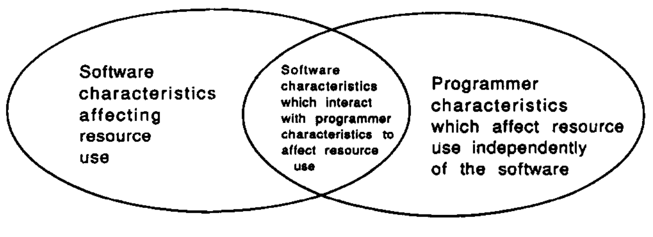
Software characteristics affecting resource use
Software characteristics which interact with programmer characteristics to affect resource use
Programmer characteristics which affect resource use independently of the software

Figure 1 Characteristics relevant to cognitive complexity

characteristics relevant to software complexity as defined in this paper.

The other advantage of Curtis' definition, which it shares with Basili's definition, is that complexity is measurable. If one includes the qualification that complexity is dependent on the task being performed, then the definition covers all the suggested ingredients of complexity. Therefore the following definition will be adopted: *The cognitive complexity of software refers to those characteristics of software which affect the level of resources used by a person performing a given task on it.*

This paper examines the complexity of software only in respect of tasks performed by programmers which involve analysing code. The tasks for which this is appropriate are manifold, including maintaining, modifying and/or extending, testing and understanding code (Figure 2). The cognition processes common to these tasks are essentially reading code (chunking) and searching through code (tracing) as discussed below under 'Cognitive complexity'.

Theoretical approaches to complexity

Existing metrics are based on a variety of different theoretical foundations (Belady[37]). It is argued here that the most appropriate approach for a model of *cognitive* complexity is to develop a framework based on research into the cognitive processes used by programmers in performing the relevant tasks. Such an approach has been endorsed by Curtis[13], Davis[38] and Kearney *et al.*[30] among others.

Figure 3 'zooms in' on the 'Software Complexity Metrics' box in Figure 2. It shows that complexity, an external characteristic or attribute, has three 'flavours': computational, psychological and representational. The most important of these is psychological (e.g. Zuse[39]) which encompasses programmer characteristics, structural complexity and problem complexity. It is often argued (e.g. Card and Glass[12], p 46) that problem complexity cannot be controlled and it is therefore frequently dismissed from consideration in the software engineering literature. On the other hand, it is important to realize that it should figure in

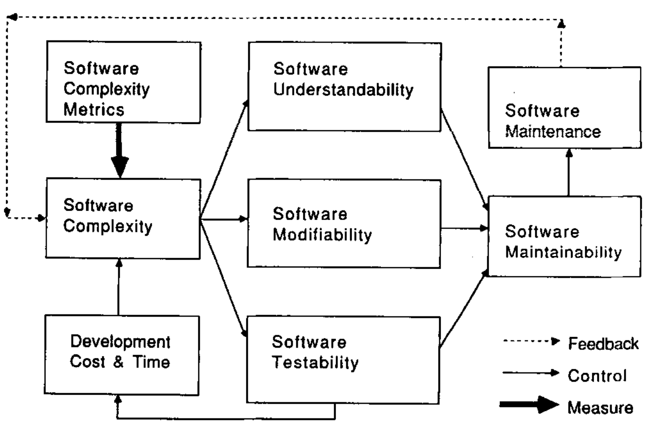

Figure 2 The interconnections between external characteristics, especially maintainability and its contributing factors (understandability, modifiability and testability) and complexity, with internal measures (of software complexity), development costs and time (after Li and Cheung[103])

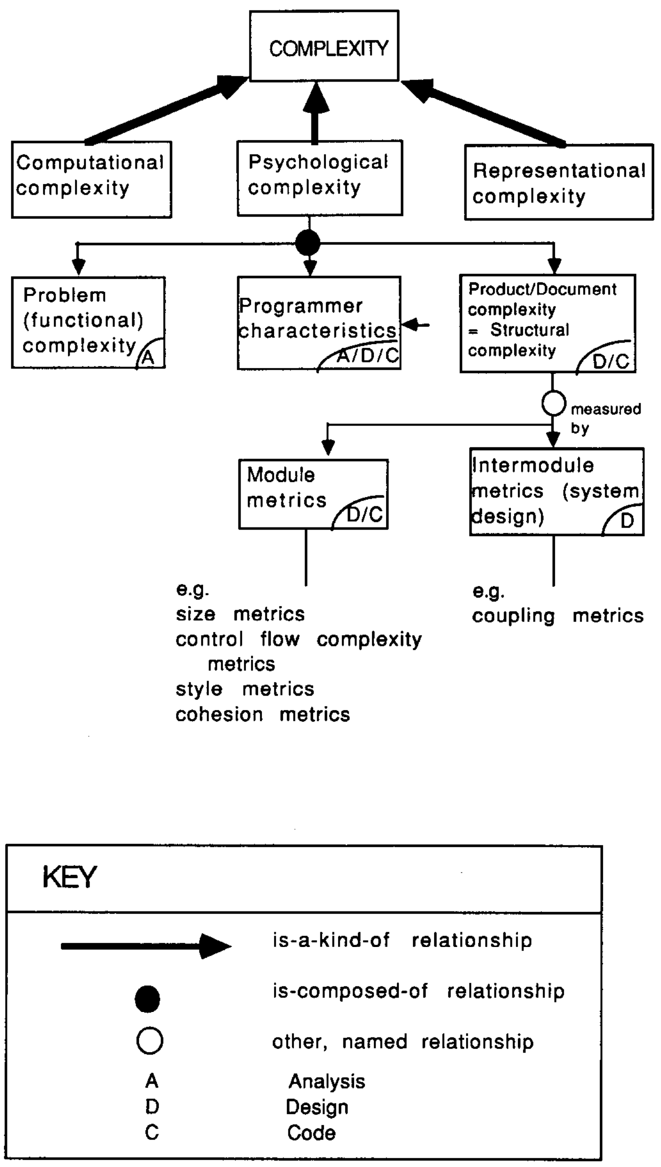
COMPLEXITY
Computational complexity
Psychological complexity
Representational complexity
Problem (functional) complexity A
Programmer characteristics A/D/C
Product/Document complexity = Structural complexity D/C
measured by
Module metrics D/C
Intermodule metrics (system design) D
e.g.
size metrics
control flow complexity metrics
style metrics
cohesion metrics
e.g.
coupling metrics
KEY
is-a-kind-of relationship
is-composed-of relationship
other, named relationship
A Analysis
D Design
C Code

Figure 3 Classification of software complexity. There are several types of complexity which are external characteristics (or attributes) or software. The most commonly discussed is structural complexity which is quantified by means of structural complexity metrics (or measures). These metrics may either be focused at the module (or intramodule) level or at the intermodule level and measure the internal attributes of the code or design.

terms of a comparison with design measures. In other words, the design and code complexity measures described above *do not* provide an absolute rating; rather they should be evaluated as relative to the problem complexity.

Most of the software engineering literature is focused on developing and validating structural complexity metrics and then using them to try to get some estimate (usually) of an external characteristic of interest (e.g. Harrison *et al.*[40])—often maintainability (Figure 2). Product or structural complexity is measured by module metrics and intermodule metrics (Kitchenham and Walker[17]). Modularity metrics are focused at the individual module level (subprogram, class) and may be measures for: (i) the internals of the module (procedural complexity) such as size, data structure or logic structure (control flow complexity); or (ii) the external specification of the module (semantic complexity)—typical module cohesion as viewed externally to the module. Intermodule metrics (measuring systems design complexity), on the other hand, describe the connections *between* modules and thus characterize the system level complexity.

Existing theoretical approaches to formulate such 'complexity' metrics include the use of information theory from signal processing (e.g. Halstead[2]; Mohanty[41]; Davis and LeBlanc[42]); approaches based on analogues with graph theory (e.g. McCabe[1]) or on lattice theory (e.g. Harrison and Magel[43]). Munson and Khoshgoftaar[44] adopt a more inductive approach, using factor analysis. These, and other, approaches are summarized in Table 1.

One feature which all of these approaches have in common is that they begin with certain characteristics of the software and attempt to determine what effect they might have on the difficulty of the various programmer tasks. A more useful approach would be first to analyse the processes involved in programmer tasks, as well as the parameters which govern the effort involved in those processes. From this point one can then deduce, or at least make informed guesses, about which code characteristics will affect these parameters. In other words, one should start with the symptoms of complexity, which are all manifested in the mind, and attempt to understand the processes which produce those symptoms. By doing so, one is more likely to discover the cause of complexity than if one simply examines the full range of possible causes blindly searching for a determining factor (see also Curtis[13]; Soloway *et al.*[45]; Curtis *et al.*[46]; Kearney *et al.*[30], p 1045).

In measurement theory terms we propose an empirical model of complexity based on the notion of cognitive complexity as the basis for the numerical model development (see Roberts[47]; Fenton[20]; Zuse[39]; Offen and Jeffery[48]). This is shown in Figure 4. The proposed empirical model of complexity provides the experimental context and a basis for experimental design, while the numerical relational model provides a formal basis for the analysis of measurement data that will result from later measurement (see Baker *et al.*[49]). In this model the essential requirement is that the numerical model preserves the basic relations that exist between the entities and the attributes in the empirical model. This duality provides the mechanism whereby the numerical relations are made meaningful in the context of the associated empirical model.

Cognitive complexity

In order to understand fully the cognitive complexity of software, the comprehension process and the limits on the capacity of the human mind must first be understood and research from cognitive science will consequently be utilized significantly in this study of software complexity metrics. Evaluation of such cognitive processes enables us to formulate a new approach to software complexity metrics.

Architecture of the mind

The mind is usually modelled by dividing it into short-term and long-term memory. The characteristics of short-term

Table 1. Taxonomy of theoretical approaches taken in existing metrics

Theoretical approach	Metric or related research
Information theoretic	Chen's[77] MIN (Nesting)
	Harrison's[78] AICC
	Halstead's[2] Effort
	Ramamurthy and Melton's[79] Weighted pair
	Hansen's[80] Complexity Pair—control flow and operators
	Davis and LeBlanc's[42] Entropy
	Woodfield *et al.*'s[81] Logical module complexity
	Gannon *et al.*'s[82] Number of generic packages
Graph theoretic	McCabe's[1] Cyclomatic number, $V(G)$
	Gill and Kemerer's[11] Complexity density
	Woodward *et al.*'s[83] Knots
	Myers'[67] Complexity interval
	Damerla and Shatz[84]
	Gilb's[85] Logical complexity
	Harrison and Magel's[43] Scope ratio (Nesting)
	Gaffney's[66] Jumps
	Adamov and Richter[86] Structural complexity
	Oviedo's[87] Branches
	Chen's[77] MIN (Nesting)
	Lakshmanan *et al.*[27]
	Ramamurthy and Melton's[79] Weighted pair
	Hansen's[80] Complexity pair—control flow and operators
	McCabe and Butler's[88] Subtree complexity
	Tsai *et al.*'s[89] Data structure complexity
	Emerson's[72] Cohesion metric
	Nejmeh's[6] NPATH Complexity
	Fenton and Whitty's[19] Structural measures
	Prather's[90] Complexity metrics based on sequence, decisions and loops
	Henderson-Sellers and Tegarden's[91] $V_{LI}(G)$ Cyclomatic metric
	Piwowarski's[92] Nesting measure
	Henderson-Sellers *et al.*'s[93] Product complexity
	Sagri's[94] Rated and operational complexity
Lattice theoretic	Harrison and Magel's[43] Scope number
Communication theory	Henry and Kafura's[23] Information flows
	Shepperd's[95] Information flows
	Bieman and Debnath's[96] Data dependency graph
Empirical	Tanik[97]
	Troy and Zweben[98]
Other	Harrison *et al.*'s[40] Span
	Munson and Khoshgoftaar's[34] Data structure complexity
	Basili and Turner's[99] Segment global usage pair
	Oviedo's[87] Reachability
	Munson and Khoshgoftaar's[7] Relative complexity
	Oviedo's[87] Weighted complexity
	Yau and Collofello's[100] Stability measure
	Tian and Zelkowitz[32]
	McClure's[101] Program complexity analysis method
	Iyengar *et al.*'s[65] Logical complexity
Cognitive science	Atwood and Ramsey's[102] Propositional analysis
	Davis and LeBlanc's[42] Review measure
	Soloway *et al.*'s[45] Plan analysis
	Bastani's[64] Complexity model

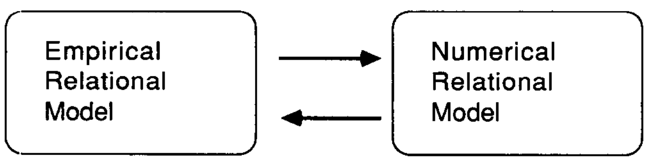

Figure 4 The essential duality in all measurement processes

memory are that distraction causes forgetting of recently learned material after about 20–30 seconds (Tracz[50], p 29; Sutcliffe[51], p 26), other simultaneous inputs impair recall, similar inputs impair recall and that recall is improved by presenting both a word and a picture together (Sutcliffe[51], p 26). The limits on the language-based sub-component of short-term memory are a relatively fast access time, a rapid loss of information unless refreshed (Sutcliffe[51], p 27), and a capacity of 7 ± 2 chunks (Miller[52]), although this capacity is reduced while performing difficult tasks (Kintsch[53]). The rate of memory decay can be slowed down by mental rehearsal of the information. Furthermore, the effective capacity of short-term memory is expanded by chunking, which involves abstracting recognized qualities from the information and storing the abstraction instead of the complete information.

Long-term memory has 'virtually unlimited' capacity and memory loss appears to be largely a problem of retrieval (Tracz[50], p 26). However, the factors that determine recall ability appear to be similar to those for short-term memory. Thus, pictures presented together with associated text tend to be remembered better than text alone. Furthermore, memory recall is retarded by extraneous 'noise', and enhanced by structure, particularly when the structure or its components already exist in long-term memory (Tracz[50]; Sutcliffe[51], pp 28–33). The structure of knowledge in long-term memory and the process of chunking involved in comprehension (a sub-task of debugging and extending) are intimately related.

The cognitive processes in comprehension

There is debate about the exact mental processes involved in debugging, extending, or simply understanding a section of code. The generally held view is that programmers '*chunk*' (Miller[52]; Shneiderman and Mayer[54]; Badre[55]; Davis[38]; Ehrlich and Soloway[56], p 114), which involves recognizing sections of code and recording the presence of that section in short-term memory by means of a single reference or memory symbol. Chunking involves recognizing groups of statements (not necessarily sequential) and extracting from them information which is remembered as a single mental abstraction. These chunks are then further chunked together into larger chunks and so on, forming a multi-levelled, aggregated structure over several layers of abstraction. However, there is also evidence that programmers perform a type of *tracing*. This involves scanning quickly through a program, either forward or backward, in order to identify relevant chunks. (For example, Weiser[57] examined 'slicing' which includes tracing backwards to ascertain the determinants of a set of variables at a point in a program). Tracing may be required to allow knowledge about certain variables or functions, scattered throughout a program, to be acquired. These two processes, of chunking and tracing, both have implications for software complexity. Furthermore, both processes are involved in the three programmer tasks of understanding, modifying and debugging.

Shneiderman and Mayer[54] suggest that the chunking process involves utilizing two types of knowledge: semantic and syntactic. Semantic knowledge includes generic programming concepts from structures as basic as loops, to more complex concepts such as sorting algorithms. This semantic knowledge is stored in a way which is relatively independent of specific programming languages. Furthermore, the information is systematically organized in a multi-levelled structure so that related concepts are aggregated into a single concept at a higher level of abstraction. Syntactic knowledge is specific to particular languages and allows semantic structures, implemented in those languages, to be recognized. By chunking, syntactic knowledge is used to convert the code into semantic knowledge, which is then stored as a multi-levelled representation of the program in memory. In a critique of this model, Gilmore and Green[58], p 463) suggest that recognition of semantic constructs may, to some extent, be dependent upon syntactic aspects of its presentation.

Ehrlich and Soloway[56] propose a more comprehensive model of chunking based on 'control flow plans' and 'variable plans'. Control plans describe control flow structures, whilst variable plans rely on a range of features common to variables with particular roles. Examples of such roles include counter variables and variables in which user input is stored. This model of chunking as a process of recognizing 'program plans', which consist of both 'control flow plans' and 'variable plans', will be adopted throughout the rest of this paper because it takes account of knowledge related to both variables and functions.

Contingency model of programmer tasks

In the range of tasks performed by a programmer, it is often necessary to locate the required chunk before comprehension can begin. A number of models of the various problem-solving tasks performed by programmers on programs, have incorporated this process of tracing. Tracing, like chunking, has a large impact on complexity.

A broad model of programmer problem solving suggested by Green *et al.*[59] (pp 223–224) takes into account a wider variety of tasks performed by programmers. In order to answer 'sequential questions' (concerning the order of events), a programmer must work forward, possibly chunking, from the given point. However, 'circumstantial questions' (concerning the conditions required for a particular action to take place) are resolved by finding a 'signpost' (i.e. some point in the program which directly causes an action under certain conditions) and working backwards from that point. Although usually related to debugging, tracing backward may also be required for both understanding and modifying. In particular, attempting to resolve a variable dependency while reading code, a programmer may have to trace backward finding references to the variable until the 'variable plan' is understood, and/or the determinants of its value ascertained. Most procedural languages tend to be 'asymmetrical' so that it is simple to trace forward, but difficult to trace backward (Green *et al.*[59], pp 223–224).

In the rest of this paper, the term 'tracing' will be used to refer to the process of searching through code in order to identify (but not comprehend) relevant chunks. The term 'chunking' will be used to refer to the process of comprehending those chunks.

A new general model of program comprehension applicable to all programmer tasks

In this section a comprehensive model of programmer problem solving is proposed, which involves a strong intertwining of the chunking and tracing processes. When a programmer is primarily chunking, there are control and variable dependencies which, in order to be resolved, require that the programmer performs a certain amount of tracing forward or backward in order to find the relevant sections of code. Having found that code, programmers will once again chunk in order to comprehend it. Conversely, when programmers are primarily tracing, they will need to chunk in order to understand the effect of the identified code fragments. For example, they may need to analyse an assignment statement in order to determine its effect, or they may need to analyse a control structure in order to understand how it interacts with the assignment statements contained in it, to create certain effects.

Keeping the interaction of these processes in mind, one can define an incremental model to describe the three programmer tasks of understanding, modifying and debugging based on chunking and tracing. This model appears in Table 2. The effects of chunking and tracing difficulty on complexity can be graphically demonstrated by modelling the various programmer tasks as 'landscapes'. Such a landscape model appears in Figure 5. The markers, at a single level, delineate each chunk. For example in Figure 5, at the top level there is a single chunk visible, delineated by the two markers (A and B); at the second level there are two chunks delineated by the two pairs of markers (C,D and D,E); and at the lowest level there is a single chunk: F,G. Note that although the chunk CD is interrupted by a lower level chunk, its integrity remains as a result of its semantic integrity. The complexity of the top level chunk is thus represented by the sum of the two line segments Ax_1 and x_4B; the overall system complexity being

Table 2. The incremental model of programming tasks

Understanding	Chunking forward for comprehension. Tracing to find related chunks in order to comprehend control and variable plans used in the immediate chunk.
Modifying	Understanding to permit modification plus tracing forward to identify ripple effects. Chunking to understand variable references and associated control structures which contribute to change propagation.
Debugging	Tracing backward to diagnose faults. Chunking to understand variable definitions and related control structures which determine the values of variables related to the faults. Modifying to correct fault.

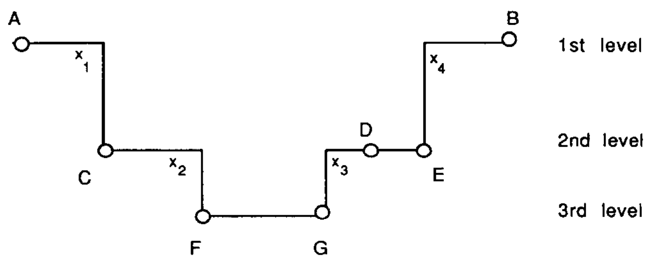

Figure 5 Landscape model of program comprehension

visualized by the total distance between the end points of the chunk: A and B.

While reading an upper level chunk, a dependency requires that the programmer suspend reading of the original code segment because of the need to undertake tracing in order to understand fully the chunk currently being analysed. The 'vertical drop' (e.g. x_1C) represents visually the work required in *tracing* the relevant code section. The length of time and amount of work required to resolve the dependency is a function of the aggregate depth of the dependency 'valley' and the aggregate breadth of the dependency 'valley'. The total *depth* of the 'nested valleys' depends on the length of the chain of dependencies which must be traced in order to satisfy the programmer's enquiry and the difficulty of performing the tracing involved in each link of the chain. In addition, the number of 'steps' involved indicates the number of chunks which need to be considered. The total *breadth* of the 'nested valleys' is determined by the effort required to understand each chunk in the dependency chain.

For example, Figure 6(a) illustrates the type of landscape which might represent the tracing of a chain of variable dependencies, either backward or forward. In contrast, Figure 6(b) illustrates the type of landscape which might represent reading a series of nested chunks. Note that the first subchunk in Figure 6(b) involves significant tracing. This could represent a called module, for example. The lowest subchunk involves almost no tracing (the minimal drop shown is to indicate that it is a nested chunk rather than a sequential chunk). This chunk is embedded within its immediate superchunk. For example, it could represent a loop nested within a module.

In mathematical terms, the difficulty of solving a programming enquiry focused on the i-th chunk is given by the complexity, C_i. This is the sum of the difficulty of

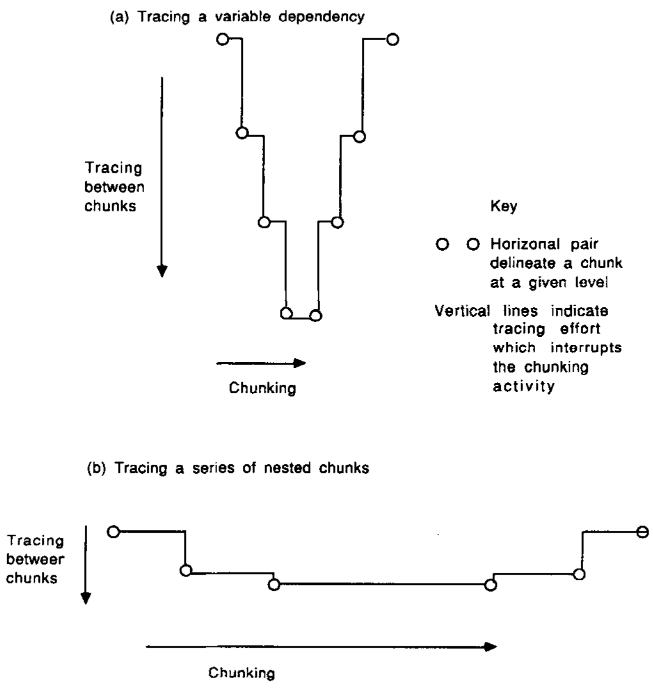
(a) Tracing a variable dependency
Tracing between chunks
Chunking
Key
O O Horizonal pair delineate a chunk at a given level
Vertical lines indicate tracing effort which interrupts the chunking activity
(b) Tracing a series of nested chunks
Tracing between chunks
Chunking

Figure 6 Landscape for a variable dependency compared with that for a series of nested chunks

understanding the immediate (i-th) chunk, R_i, and the difficulty in moving from the original top-level (the whole program) down to the current (i-th) chunk. That difficulty occurs from both chunking and tracing, summed over all the levels between the top level and the current chunk (as seen in Figure 5). For example, in Figure 5, to understand chunk FG (R_i) we need to understand chunk CD (C_{CD}) and to trace from CD to FG (T_{CD} represented by 'depth' x_2F or x_3G). Recursively, then, to understand CD we need to also understand AB (C_{AB}) and trace from AB to CD viz. x_1C (T_{AB}). Thus total complexity related to understanding FG is represented by

$$C_i = R_i + C_{CD} + T_{CD} + C_{AB} + T_{AB} \qquad (1)$$

Generalizing this over the full set of chunks, denoted by N, on which the i-th chunk is directly dependent for a given task

$$C_i = R_i + \sum_{j \in N} C_j + \sum_{j \in N} T_j \qquad (2)$$

where T_j is the difficulty of tracing a particular dependency.

Clearly Equation (2) is a recursive definition so that for most levels of nesting, C_i will be partly determined by C_j. At the lowest level of nesting C_i will be equal to R_i since there will be no more nested chunks. For a program with only one chunk, $C_1 = R_1$ as expected. Factors determining R_i, N and T_i will be discussed below.

Operationalizing the cognitive complexity model

In order to operationalize the Cognitive Complexity Model (CCM), as defined by Equation (2), we need to understand what characteristics of the code and the programmer are relevant to quantifying these processes of chunking and tracing. Analysis of each of these processes in turn leads to the identification of independent variables which can be measured. Each of these must be shown to be valid in terms of both measurement theory and practicality—for instance, requiring extensive data collection and statistical and theoretical analysis. In the sections that follow, we make no attempt to categorically identify these necessary components of the CCM; rather we identify the conceptual concerns necessary for identification of these component measures and, when available, draw *one illustrative example* from the literature. The aim here is more to focus on a research agenda likely to provide useful results than to support any particular 'brand' of component metric.

The definition of a chunk

It is difficult to determine exactly what constitutes a chunk since it is a product of a programmer's semantic knowledge, as developed through experience. However, generalizations can be made in order to operationalize the construct encapsulated in Equation (2). For example, Mayer[60] classified chunks as 'mandatory' or 'non-mandatory'. Mandatory chunks are those in which one statement must always be followed by a second statement. For example, in certain languages, a FOR must always be followed by a NEXT, and an IF must always be followed by a THEN and an ENDIF. Non-mandatory chunks are those which a programmer uses for convenience, such as search loops. In this paper, for purposes which only involve normal chunking forward, the corresponding delimeters of each control construct (e.g. loop, conditional) will be used to distinguish each chunk. One special case which will be defined is where a routine contains a single loop or decision structure, at the next level of nesting. In this case, the code surrounding the decision or loop structure is often simply associated with initialization, and therefore the whole structure should be considered as one chunk.

This definition of a chunk is inadequate when describing processes which primarily involve tracing. Where a programmer is tracing a variable forward or backward, each chunk would often be a single statement containing a variable definition. Therefore when considering tracing, the instruction line containing the definition should be regarded as a chunk. Again, the exception to this is where the variable definition is contained within a conditional control structure, or a series of nested conditional control structures. Under such circumstances, all of the conditional control structures containing the variable definition should be regarded as relevant and divided into chunks in the normal way.

Chunks may be classified as elementary or compound. Elementary chunks consist only of sequentially self-contained statements. Compound chunks are those which contain within them other chunks. Routines containing a number of procedure calls should be considered compound chunks rather than elementary chunks, since the identifier of each procedure is usually a cue to some abstract programming unit which is itself a chunk. At the same time, routines containing no routine calls may also be compound chunks. If a routine contains more than one recognizable subunit, it is equivalent to a module containing many routine calls in the sense that both contain within them multiple subchunks. This conforms with the notion underlying the model of comprehension described earlier, that a loop or decision structure can be considered a nested dependency in much the same way as can a procedure call.

Factors affecting R*: the complexity of the immediate chunk*

Both elementary and compound chunks are influenced by a range of factors determining the complexity of the immediate chunk, some of which were shown explicitly in Figure 3. They include: (1) size (R_S); (2) the difficulty of comprehending the control structure within which the chunk is contained (R_C); (3) the difficulty of comprehending Boolean expressions contained in the chunk (R_E); (4) the recognizability (R_R); (5) the visual structure/layout of the program (R_V); (6) disruptions caused by dependencies of those chunks (R_D); and (7) familiarity (R_F) which influences speed of recall, i.e.

$$R = R(R_S, R_C, R_E, R_R, R_V, R_D, R_F) \tag{3}$$

Each of these will be reviewed in the following seven subsections in which some tentative proposals for possible submodels will be made, whilst recognizing that there remain a significant number of good alternatives. Selection of these alternatives may indeed be task-dependent (e.g. maintainability cf. comprehensibility: Basili and Rombach[61]; Fenton[62]) and require extensive evaluation and validation—a topic well beyond the scope of this paper, but a project already under way (Cant *et al.*[63]).

Chunk size (R_S). There are two issues to the size question. The first is the structural complexity of the actual chunk, as read by the programmer, resulting from its size, as exemplified by lines of code. The second is the psychological complexity of identifying a chunk where a long contiguous section of non-branching code must be divided up in order to be understood by the programmer.

A possible equation describing the effect of the size of any chunk is therefore proposed:

$$R_S = \begin{cases} aS_i & \text{if } S_i \le L_{max} \\ aS_i + b\left(\frac{S_i - L_{max}}{L_{max}}\right) & \text{if } S_i > L_{max} \end{cases} \tag{4}$$

where R_S is the complexity resulting from chunk size, S_i is the size of the chunk, L_{max} is some limit on the size of a chunk which may be estimated empirically and a, b are empirically determined coefficients. One difficulty may be that the limit, L_{max}, would probably vary from programmer to programmer.

The type of control structure in which it is enclosed (R_C). Intuitively, a conditional control structure is more complex to understand than a normal sequential section of code. However, it is also highly likely that different levels of complexity are associated with different types of control structures. In order to understand a decision structure, a programmer must first understand the Boolean expression, and then analyse the distinction between the alternative code. In order to understand a loop, a programmer must 'inductively' develop an understanding of the effect of the loop. If the function of the loop is not immediately recognized, then a programmer must symbolically iterate the function of the code contained in the loop in order to follow how the values of variables contained in the loop change. This distinction between loops and decision structures is supported by Bastani[64], Iyengar *et al.*[65], Evangelist[5] and Nejmeh[6]. This distinction is not included in McCabe's cyclomatic complexity, $v(G)$. Although this has led to a number of criticisms (e.g. Evangelist[5]; Nejmeh[6]), McCabe's[1] original intention was to provide a measure of decisions for use in the development of testing procedures.

For example, equations for R_C are given by Iyengar *et al.*[65] and Bastani[64], the latter emphasizing the iterative nature of comprehension of loop structures, suggesting that

$$R_C = \sum_{j=1} C_i^{(j)} p^{2^{j-1}-1} \tag{5}$$

where $C_i^{(j)}$ is the complexity of the i-th chunk after the j-th iteration and p is the probability of failure after the first iteration.

Empirical evidence for the usefulness of these, and other, metrics for R_C remains indecisive (e.g. Gaffney[66]).

The difficulty of understanding complex Boolean and other expressions (R_E). Boolean expressions are an important issue associated with control flow structures. There is very little understanding of the impact of control structures on complexity. McCabe[1] assumes that each conditional within a Boolean expression adds the same level of difficulty as an entire control structure. A similar assertion is made by Nejmeh[6]. Such a treatment of conditionals effectively ignores the debate about whether long Boolean expressions are more or less complex than extended sequences of nested conditional structures.

Myers[67] argues that a series of conditionals contained in a Boolean expression is considerably simpler than the equivalent array of nested control structures. Green *et al.*[59] (p 225) suggested that complex Boolean expressions would significantly increase the mental workload required to understand a chunk, thereby increasing the likelihood of mistakes. Evangelist[5] (pp 390–391) points out that logical expressions are no more difficult to explicate than normal arithmetic operations. He claims that the complexity of Boolean expressions needs to be determined empirically. However, the exact effect of Boolean expressions on psychological complexity is not well understood. A simple summation provides the simplest acceptable model which, based on the principle of scientific parsimony, can be proposed as

$$R_E = b_1 \sum_{\substack{\text{all Boolean} \\ \text{expressions}}} B_i \tag{6}$$

where B_i is the number of predicates in the i-th Boolean and b_1 an empirical constant. A formula such as Equation (6) is purely tentative at this stage and provides an indicator of a useful area for further research.

Despite the obvious importance of this issue it is by no means settled. The most useful approach is probably to highlight both excessively complicated sequences of nesting, and also overly complicated Boolean expressions. In order to ensure that the model proposed in this paper is consonant with this approach, long Boolean expressions should be taken into account. The effect of deep sequences of nested conditionals is already accounted for by the level of disruption (see later, below) to the comprehension of superchunks (outer conditional structures). Empirical evaluation of R_E thus remains an open question.

The recognizability of a chunk (R_R). Soloway and Ehrlich[68] have argued that complexity is largely dependent on qualitative factors, including programming experience, which determine how easily a chunk is recognized. They conducted experiments which demonstrated that 'programming plans' are not recognized by experienced programmers when the 'rules of discourse' are not followed. For example, programmers will find it significantly harder to recognize a routine to find the minimum value from an input stream if the variable containing the result is called MAX. Furthermore, programmers will not recognize structures requiring a conditional, if a 'while' statement is used where a simple 'if' would suffice. This theory is supported by Gilmore and Green's[58] (p 463) experiment which showed that the mental abstraction of a program was highly dependent on the syntactic form of the program.

Cohesion may also have an important impact on the recognizability of a chunk. The importance of cohesion was identified by Yourdon and Constantine[69]. Cohesion refers to the 'degree to which the tasks performed by a single program module are functionally related' (IEEE[70]). This concept may be applied to a chunk, as defined in this paper, in order to determine to what extent functions performed by a chunk are related. Where a chunk contains two or more functions, related only by the same control structure, it is likely that the chunk will be significantly harder to recognize.

These two factors (conformance with rules and cohesion) may be quantified separately

$$R_R = r_R + r_C \tag{7}$$

where r_R and r_C represent the complexity of these two factors, respectively. The value of the factor r_R may be described by Moher's[71] metric which gives

$$r_R = -\log_2 \left[\prod_{k=1}^{N} \mathrm{P}(t_k) \right] \tag{8}$$

where $\mathrm{P}(t_k)$ is the probability of the k-th token being drawn from a vocabulary of n tokens specified within the 'rules of discourse'. However, it should be noted that a significant difficulty in deriving values for r_R using Equation (8) is the enormous task of compiling the large number of rules of discourse against which to judge the expectedness of any particular token.

The second factor, cohesion, as originally proposed by Yourdon and Constantine[69], has seven different levels; although only three groups are used by Emerson[72] who gives*

$$r_C = \frac{|M_i| \dim M_i}{|VF - \{T\}| \dim VF} \tag{9}$$

where the set of nodes, VF, in a directed graph, F, contains a subset $\{M_i\}$, and T is the terminal vertex. The metric r_C then gives the cohesion of this module to the average cohesion of the reference sets using Equation (9) or an acceptable alternative.

The effects of visual structure (R_V). A related issue is the visual structure provided by a program. Green *et al.*[59] (pp 225, 227–228) note that chunks which are visible and tractable (i.e. separable from the rest of the program) considerably reduce the difficulty of comprehending a chunk. Note, however, that natural chunks may be larger for experienced programmers, particularly where they have experience with the program and/or language under consideration.

There are essentially three levels of chunk identification difficulty, common to most programs:

(1) Chunks delineated by routine boundaries are the easiest to identify, especially if well named.

*Upon correcting typographical errors in the source.

(2) Chunks delineated by control structure boundaries only are less easily identified since there are less cues provided concerning their function, and the chunk is not as obviously separated from the surrounding code.
(3) Chunks not delineated by any boundaries are the most difficult to identify. One of the reasons that a chunk may be in this category is because a chunk in one of the higher categories becomes so large that it can no longer be understood as a single unit. The point at which this occurs will differ widely from programmer to programmer and from chunk to chunk making it difficult to quantify the process.

Hence it is reasonable to model these effects, at their simplest, as

$$R_V = a_1 V \tag{10}$$

where $V \in \{1,2,3\}$ represents the three levels of identification difficulty outlined above and a_1 is an empirical constant which still requires evaluation experimentally.

Disruptions in chunking caused by dependencies (R_D). In the process of understanding a chunk, dependencies often arise such that knowledge of some other 'variable plan' and/or 'control flow plan' is needed in order to understand that chunk fully. Often the programmer may recall the necessary program plans from memory. However at other times the dependencies must be resolved by reviewing the relevant sections of code. This often involves tracing the relevant sections and then chunking them in order to acquire understanding. Resolving dependencies, in this manner, disrupts the comprehension of the original chunk. Since short term memory has only limited capacity, and a rapid decay rate which is worsened by distraction (Sutcliffe[51], p 26), disruptions to the chunking process may cause knowledge of the original chunk to be lost.

This disruption effect is not simply a result of the effort involved in tracing, but also a result of the effort in understanding the nested chunks. Therefore, the effect of disruption is applicable not only to remote dependencies (a point emphasized in the empirical study of Cant *et al.*[63] requiring tracing, but also applicable to embedded dependencies, such as nested loops or decision structures, which must be understood before the chunk within which it is nested may be understood.

A possible equation for the disruption effect, R_D, of set of chunks N, and their descendants, on the i-th chunk is

$$R_D = d \sum_{j \in N} C_j + e \sum_{j \in N} T_j \tag{11}$$

where R_D is that part of the complexity of a chunk resulting from disruptions by nested chunk, e.g. subroutine calls; N is the set of chunks on which the i-th chunk is directly dependent for a given programmer task, T is the difficulty of tracing a particular dependency, and d and e are empirically determined constants.

Speed of review or recall (R_F). The ability of a programmer to understand a chunk is often affected by the number of times that the programmer has read that particular chunk on previous occasions. This may be referred to as chunk familiarity. Chunk familiarity may be looked at from two aspects. The first is by considering the number of times that a programmer has had to review a chunk in order to understand the section of code which is being worked on in a particular session. The second is by considering the experience that the programmer has had generally with the relevant chunk or with the program in general.

Woodfield[73] (see also discussion of Davis[38]) found that repeated reviews of the same chunk reduced the amount of time required to understand that chunk. A good approximation of this effect involves assuming that each review takes two-thirds of the time of the previous review. Woodfield[73] then derives an equation for program complexity in which the chunk complexity is expressed as

$$C_i = \sum_{m=0}^{(\text{fan-in})_i} R_i f^m \tag{12}$$

where R_i is the complexity of the i-th chunk, equated in Woodfield's analysis simply to the number of lines of code in the i-th chunk. The factor f is a review constant $\sim \frac{2}{3}$. The summation over m represents the number of chunks affected by a particular chunk. Thus we can revise this for our cognitive complexity model as a multiplicative recall factor, R_F, for the i-th chunk (R_{Fi}) given as

$$R_{Fi} = \sum_{j \in N} f^j \tag{13}$$

for use in Equation (3)—and hence in Equation (2).

Operationalizing R. The terms in Equation (3), based on the above discussion, would appear to be likely to be generally addictive (with the exception of R_F which is clearly multiplicative). Empirical research is still required to validate this hypothesis and is the next stage in our research program (Cant *et al.*[63]).

Operationalizing the dependency construct

'Dependency' in a chunk may refer to any part of a program which must be understood in order to determine the function or effect of a chunk, or of a part of a chunk. Therefore dependencies may be functional dependencies or variable dependencies. Functional dependencies refer to procedures or functions called from within a chunk, which must, as a result, be read in order to understand that chunk. Variable dependencies may refer not only to definitions which may affect the variable reference under consideration, but also to variable references which may be affected by a variable definition currently under consideration. These will be distinguished by calling the former 'determinants' and the latter 'effects'.

Dependencies may also be embedded, local or remote. Remote dependencies are those which extend across modular boundaries. Local dependencies are contained within these modular boundaries (but for our purposes, not within the chunk concerned). Embedded dependencies are physically contained within the chunk itself.

Factors affecting the number of dependencies traced, N. The number of actual dependencies may be determined by identifying all the dependencies of each of the various types. However, the relevant dependencies depend on the programmer task being considered. In understanding a chunk, variable 'determinants' and functional dependencies will be relevant; whereas in tracing ripple effects, variable 'effects' will be relevant and in isolating errors, variable 'determinants' will be relevant. Remote variable 'determinants' may be parameters passed to the module containing the chunk, values returned by called procedures or global variables referenced in the chunk. Similarly, remote variable 'effects' may arise from definitions of global variables, parameters passed to called procedures or from values returned to the calling module.

Local variable 'determinants' and 'effects' may occur before or after the chunk because of loops within which the chunk may be embedded. Note also that variable dependencies may be indirect, in the sense that the value of a variable may depend on a control structure (decision or loop) and therefore upon the variables which control the execution of that control structure.

For embedded functional dependencies such as nested loops or nested decision statements, it can generally be assumed that they are always read. The corollary of this is that variable dependencies contained within the same chunk will usually not require tracing, because they have been read recently, or will be read shortly after reading the program statement concerned. Although these assumptions are not necessarily always valid, they are found to be generally applicable and allow simplifications to be made which may be useful for developing a metric.

When simply reading for understanding, dependencies are only traced to the extent necessary to give a general idea about the 'control plan' or 'variable plan' of the particular module or variable referenced. Therefore one factor which bears on whether a particular dependency will be traced depends on the depth already traced and the depth of understanding required.

Another factor which affects whether a dependency will be traced is the level of cues in the program text which help recall of the relevant control flow or variable 'plans'. This is supported by Woodfield *et al.*[74] who found that the use of comments within the code significantly increased the level of programmer comprehension. The only difficulty with comments and other cues is that they are difficult to analyse automatically. Therefore measures of the level of cues provided will either need to be subjectively based or based on heuristics.

When reading a program in order to trace the ripple effects of a change, 'effects' must be traced until the programmer is satisfied that the change will not be propagated beyond that point. Similarly, when reading a program in order to isolate a bug, 'determinants' must be traced until the bug is found.

Factors affecting T, *the difficulty of tracing*

The difficulty of tracing is a function of many factors including the localization of the dependencies (T_L), the ambiguity of the dependency (T_A), the spatial dependency of the dependency (T_S), the level of cueing of the dependency (T_C) and the familiarity of the dependency (T_F).

$$T = T(T_L, T_A, T_S, T_C, T_F) \tag{14}$$

Localization (T_L). This is the degree to which a dependency may be resolved locally. The various levels are embedded, local and remote. The simplest equation might be:

$$T_L = a_2 L \tag{15}$$

where $L \in \{1,2,3\}$ represents the three levels of embedded, local and remote and a_2 is an empirical coefficient requiring empirical evaluation.

Ambiguity (T_A). Where there are several alternative chunks on which a section of code may be dependent, or which may be affected by a section of code, the ambiguity increases the complexity of tracing the alternative chunks since, because there is no specific limit to the number of alternatives, then for any particular alternative there is no indication that it actually exists. For example, one may be uncertain about how many references there are to a global variable. Because of this ambiguity one must continue searching even when all the references have been found (cf. Dunsmore and Gannon[75]). The same is true of tracing global variables definitions, and tracing calls to a procedure. An equation might be

$$T_A = a_3 A \tag{16}$$

where $A \in \{0, 1\}$. Here a value of 1 represents ambiguity and 0 no ambiguity.

Spatial distance (T_S). This is factor representing the distance, ΔS, possibly in lines of code, between two chunks for which there is a dependency. It can be expressed simply as

$$T_S = b_2 \Delta S \tag{17}$$

where b_2 is an empirical constant. It is conceivable that this would affect the difficulty of tracing.

Level of cueing (T_C). When searching for a procedure, its name is usually found at the beginning. When searching for procedure calls, they are often obscured by being embedded within the text. Other cueing difficulties may be identified for specific languages.

$$T_C = a_4 B \tag{18}$$

where $B \in \{0, 1\}$. Here $B = 1$ represents the existence of obscure references.

Dependency familiarity (T_F). Dependency familiarity is similar to chunk familiarity and would be incorporated into Equation (14) as a multiplicative factor.

$$T_{Fi} = \sum_{j \in N} f^j \tag{19}$$

Operationalizing the tracing difficulty. Empirical research on the functional form of the tracing difficulty, T, as given by Equation (14), still remains to be undertaken. If the simple binary representation for T_A and T_C are appropriate, then once the weightings have been determined, the simplest likely representation for T would be

$$T = (T_L + T_A + T_S + T_C)T_F \qquad (20)$$

Testing these formulae constitute the next phase of the project, with special emphasis on the complexity of object-oriented software (Cant *et al.*[63]).

Conclusions and further research

A conceptual model of cognitive complexity based on an analysis of the programming process, rather than the program product, has been proposed. This model takes two forms: a graphical representation via the landscape diagram, and a mathematical representation.

The landscape diagram is approximate and useful for human judgement processes concerned with a rough appreciation of the cognitive complexity of code for a defined programming purpose. The mathematical expression of cognitive complexity given is, as yet, also a conceptual model requiring considerable work before a fully operational version can be provided. Only tentative formulae for describing the constituent parts of chunking and tracing have been discussed, purely as material illustrative of the underlying concepts. The complexity model is designed to be useful in all programming paradigms and current research is aimed at exploring its operational applicability to the object-oriented paradigm, initially in terms of the significant effect of remote dependencies. For example, Cant *et al.*[63] have showed that for object-oriented programs written in C++ an evaluation of dependencies did suggest agreement between perceived and predicted values in three contexts: understanding, tracing modifications and effort to isolate errors.

The model proposed here defines the meaning of cognitive complexity and, based on this, provides metrics of this complexity which can be used for benchmarking, prediction and control once the necessary empirical work has been carried out to establish the efficacy of the model. Some preliminary case study-based empirical work in Cant[76] and for C++ programs in Cant *et al.*[63] shows initial support for the model proposed.

References

1 McCabe, T J 'A complexity measure' *IEEE Trans. Soft. Eng.* Vol 2 No 4 (1976) pp 308–320

2 Halstead, M H *Elements of software science* Elsevier/North-Holland (1987)

3 Coulter, N S 'Software science and cognitive psychology' *IEEE Trans. Soft. Eng.* Vol 9 No 2 (1983) pp 166–171

4 Shen, V Y, Conte, S D and Dunsmore, H E 'Software science revisited: a critical analysis of the theory and its empirical support' *IEEE Trans. Soft. Eng.* Vol 9 No 2 (1983) pp 155–165

5 Evangelist, M 'An analysis of control flow complexity' *COMPSAC '84* (1984) pp 388–396

6 Nejmeh, B A 'NPATH: a measure of execution path complexity and its applications' *Comm. ACM* Vol 31 No 2 (1988) pp 188–200

7 Munson, J C and Khoshgoftaar, T M 'Applications of a relative complexity metric for software project management' *J. Systems and Software* Vol 12 (1990) pp 283–291

8 Munson, J C and Khoshgoftaar, T M 'Measuring dynamic program complexity' *IEEE Software* (November 1992) pp 48–55

9 Shepperd, M 'A critique of cyclomatic complexity as a software metric' *Software Engineering J.* Vol 3 (1988) pp 30–36

10 Bowman, B J and Newman, W A 'Software metrics as a programming training tool' *J. Systems and Software* Vol 13 No 2 (1990) pp 139–147

11 Gill, G K and Kemerer, C F 'Cyclomatic complexity density and software maintenance productivity' *IEEE Trans. Soft. Eng.* Vol 17 No 12 (1991) pp 1284–1288

12 Card, D N and Glass, R L *Measuring software design quality* Prentice-Hall (1990) p 129

13 Curtis, B 'In search of software complexity' in *Workshop on quantitative software models* IEEE (1979) pp 95–106

14 McCall, J A, Richards, P G and Walters, G F *Factors in software quality* Vols I, II and III (1977) (NTIS AD/A-049 014/015/055)

15 Boehm, B W, Brown, J R, Kaspar, H, Lipow, M, Macleod, G J and Merritt, M J *Characteristics of software quality* (1978) North-Holland

16 Bowen, T P, Wirgle, G B and Tsai, J *Specification of software quality attributes* Vols I, II and III (1984) (D182-11678-1/2/3). Prepared for Boeing RADC

17 Kitchenham, B A and Walker, J G 'The meaning of quality' *Procs. Conf. Soft. Eng. 86* (1986) pp 393–406

18 Fenton, N and Kaposi, A 'Metrics and software structure' *Inf. and Soft. Technol.* Vol 29 No 6 (1987) pp 301–320

19 Fenton, N E and Whitty, R W 'Axiomatic approach to software metrication through program decomposition' *Computer J.* Vol 29 No 4 (1986) pp 329–339

20 Fenton, N *Software metrics: a rigorous approach* Chapman & Hall (1991)

21 Fenton, N and Melton, A 'Deriving structurally based software measures' *J. Systems and Software* Vol 12 (1990) pp 177–187

22 Fenton, N 'When a software measure is not a measure' *Software Engineering J.* (1992) pp 357–362

23 Henry, S and Kafura, D 'Software structure metrics based on information flow' *IEEE Trans. Soft. Eng.* Vol 7 No 5 (1981) pp 510–518

24 Zuse, H and Bollman, P 'Software metrics: using measurement theory to describe the properties and scales of static software complexity metrics' *Sigplan Notices* Vol 24 No 8 (1989) pp 23–33

25 Henry, S and Salig, C 'Predicting source-code complexity at the design stage' *IEEE Software* Vol 7 No 2 (1990) pp 36–44

26 MacDonnell, S G 'Rigor in software complexity measurement experimentation' *J. Systems and Software* Vol 16 (1991) pp 141–149

27 Lakshmanan, K B, Jayaprakash, S and Sinha, P K 'Properties of control flow complexity measures' *IEEE Trans. Soft. Eng.* Vol 17 No 12 (1991) pp 1289–1295

28 Khoshgoftaar, T M, Munson, J C, Bhattacharya, B B and Richardson, G D 'Predictive modeling techniques of software quality from software measures' *IEEE Trans. Soft. Eng.* Vol 18 No 11 (1992) pp 979–987

29 Mata-Toledo, R A and Gustafson, D A 'A factor analysis of software complexity measures' *J. Systems and Software* Vol 17 (1992) pp 267–273

30 Kearney, J K, Sedlmeyer, R L, Thompson, W B, Gray, M A and Adler, M A 'Software complexity measurement' *Comm ACM* Vol 29 No 11 (1986) pp 1044–1050

31 Weyuker, E J 'Evaluating software complexity measures' *IEEE Trans. Soft. Eng.* Vol 14 No 9 (1988) pp 1357–1365

32 Tian, J and Zelkowitz, M V 'A formal program complexity model and its application' *J. Systems and Software* Vol 17 (1992) pp 253–266

33 Cherniavsky, J C and Smith, C H 'On Weyuker's axioms for software complexity measures' *IEEE Trans. Soft. Eng.* Vol 17 No 6 (1991) pp 636–638

34 Munson, J C and Khoshgoftaar, T M 'Measurement of data structure complexity' *J. Systems and Software* Vol 20 (1993) pp 217–225

35 Klir, G J *Architecture of systems problem solving* Plenum Press (1985) pp 325–353

36 Basili, V R 'Qualitative software complexity models: a summary' in *Tutorial on models and methods for software management and engineering* IEEE Computer Society Press, Los Alamitos, CA (1980)

37 Belady, L A 'On software complexity' *Workshop on Quantitative Software Models* IEEE, New York (1979) pp 90–94

38 Davis, J S 'Chunks: a basis for complexity measurement' *Inf. Proc. and Management* Vol 20 No 1 (1984) pp 119–127

39 Zuse, H *Software complexity: measures and methods* DeGruyter (1991)

40 Harrison, W, Magel, K, Kluczny, R and DeKnock, A 'Applying software complexity metrics to program maintenance' *IEEE Computer* (September 1982) pp 65–79

41 Mohanty, S N 'Models and measurements for quality assessment of software' *Computing Surveys* Vol 11 No 3 (1979) pp 251–275

42 Davis, J S and LeBlanc, R J 'A study on the applicability of complexity measures' *IEEE Trans. Soft. Eng.* Vol 14 No 9 (1988) pp 1366–1371

43 Harrison, W and Magel, K 'A topological analysis of computer programs with less than three binary branches' *ACM SIGPLAN Notices* (April 1981) pp 51–63

44 Munson, J and Khoshgoftaar, T 'The dimensionality of program complexity' *Proc. 11th Ann. Int. Conf. Software Engineering* Pittsburgh (1989) pp 245–253

45 Soloway, E, Ehrlich, K and Black, J B 'Beyond numbers: don't ask "How Many"...ask "Why"' in Janda, A (ed) *Human Factors in Computing: Proc. of CHI '83 Conf.* (1983) pp 240–246

46 Curtis, B, Soloway, E M, Brooks, R E, Black, J B, Ehrlich, K and Ramsey, H R 'Software psychology: the need for an interdisciplinary program' *Proc. IEEE* Vol 74 No 8 (1986) pp 1092–1106

47 Roberts, F S 'Measurement theory' in *Encyclopedia of mathematics and its applications* Vol 7 (1979) Addison-Wesley

48 Offen, R and Jeffery, D R 'A model-based approach to establishing and maintaining a software measurement program' *ITRC Research Report* UNSW (September 1994)

49 Baker, A L, Bieman, J M, Fenton, N, Gustafson, D A, Melton, A and Whitty, R 'A philosophy for software measurement' *J. Systems and Software* Vol 12 (1990) pp 277–281

50 Tracz, W J 'Computer programming and the human thought process' *Software Practice and Experience* Vol 9 (1979) pp 127–137

51 Sutcliffe, A *Human-computer interface design* Springer-Verlag (1989)

52 Miller, G A 'The magic seven plus or minus two. Some limits on our capacity for processing information' *Psychological Review* Vol 63 (1956) pp 81–97

53 Kintsch, W *Memory and cognition* John Wiley (1977)

54 Shneiderman, B and Mayer, R 'Syntactic/semantic interactions in programmer behaviour: a model and experimental results' *Int. J. Computer and Information Sciences* Vol 8 No 3 (1979) pp 219–238

55 Badre, A 'Designing chunks for sequentially displayed information' in Badre, A and Shneiderman, B (eds) *Directions in human computer interaction* Ablex Publishing (1982) pp 179–193

56 Ehrlich, K and Soloway, E 'An empirical investigation of the tacit plan knowledge in programming' in Thomas, J C and Schneider, M L (eds) *Human factors in computer systems* Ablex Publishing (1984) pp 113–133

57 Weiser, M 'Program slicing' *Proc. 5th Int. Conf. SE* (1981) pp 439–449

58 Gilmore, D J and Green, T R G 'The comprehensibility of programming notations' in Shackel, B (ed) *Human-computer interaction—INTERACT '84* IFIP (1985) pp 461–464

59 Green, T R G, Sime, M E and Fitter, M J 'The art of notation' in Coombs, M J and Alty, J L (eds) *Computing skills and the user interface* Academic Press (1981) pp 221–251

60 Mayer, R E 'A psychology of learning BASIC' *Comm ACM* Vol 22 No 11 (1979) pp 589–593

61 Basili, V R and Rombach, H D 'The TAME project: towards improvement oriented software environments' *IEEE Trans. Soft. Eng.* Vol 14 No 6 (1988) pp 758–773

62 Fenton, N E 'Software assessment: a necessary scientific basis' *IEEE Trans. Soft. Eng.* Vol 20 No 3 (1994) pp 199–206

63 Cant, S N, Henderson-Sellers, B and Jeffery, D R 'Application of cognitive complexity metrics to object-oriented programs' *J. Obj.-Oriented Programming* Vol 7 No 4 (1994) pp 52–63

64 Bastani, F B 'An approach to measuring program complexity' *COMPSAC '83* (1983) pp 1–8

65 Iyengar, S S, Parameswaran, N and Fuller, J 'A measure of logical complexity of programs' *Computing Language* Vol 7 (1982) pp 147–160

66 Gaffney, J E 'Program control complexity and productivity' *IEEE Trans. Soft. Eng.* Vol 10 No 4 (1979) pp 459–464

67 Myers, G J 'An extension to the cyclomatic measure of program complexity' *SIGPLAN Notices* (October 1977) pp 61–64

68 Soloway, E and Ehrlich, K 'Empirical studies of programming knowledge' *IEEE Trans. Soft. Eng.* Vol 10 No 5 (1984) pp 595–609

69 Yourdon, E and Constantine, L L *Structured design: fundamentals of a discipline of computer program and systems design*, Yourdon Press/Prentice-Hall (1979)

70 IEEE, IEEE Standard 729-1983 *IEEE Standard Glossary of Software Engineering Terminology* IEEE, NY (1983)

71 Moher, T C 'Estimating the distribution of software complexity *within* a program' in Curtis, B and Borman, L (eds) *Proc. CHI '85: Human Factors in Computing* ACM (1985) pp 61–64

72 Emerson, T J 'A discriminant metric for module cohesion' *Int. Conf. SE 1984* (1984) pp 294–303

73 Woodfield, S N *Enhanced effort estimation by extending basic programming models to include modularity effects* PhD Thesis (1980) Dept. Computer Science, Purdue University

74 Woodfield, S N, Dunsmore, H E and Shen, V Y 'The effect of modularization and comments on program comprehension' *Proc. 5th Int. Conf. SE 1981* (1981a) pp 215–223

75 Dunsmore, H E and Gannon, J D 'Analysis of the effects of programming factors on programming effort' *J. Systems and Software* (1980) pp 141–153

76 Cant, S N *The cognitive complexity of programs developed using the object-oriented paradigm* Honours Thesis (1991) School of Information Systems, University of New South Wales

77 Chen, E T 'Program complexity and programmer productivity' *IEEE Trans. Soft. Eng.* Vol SE 4 No 3 (1978) pp 187–194

78 Harrison, W 'An entropy-based measure of software complexity' *IEEE Trans. Soft. Eng.* Vol 18 No 11 (1982) pp 1025–1029

79 Ramamurthy, B and Melton, A 'A synthesis of software science measures and the cyclomatic number' *Trans. Soft. Eng.* Vol 14 No 8 (1988) pp 1116–1121

80 Hansen, W J 'Measurement of program complexity by the pair (cyclomatic number, operator count)' *ACM SIGPLAN Notices* (April 1978) pp 29–33

81 Woodfield, S N, Shen, V Y and Dunsmore, H E 'A study of several metrics for programming effort' *J. Systems and Software* Vol 2 (1981b) pp 97–103

82 Gannon, J D, Katz, E E and Basili, V R 'Metrics for Ada packages: an initial study' *Comms. ACM* Vol 29 (1986) pp 616–623

83 Woodward, M, Hennell, M and Hedley, D 'A measure of control flow complexity in program text' *IEEE Trans. Soft. Eng.* Vol 5 No 1 (1979) pp 45–50

84 Damerla, S, and Shatz, S M 'Software complexity and Ada rendezvous: metrics based on non-determinism' *J. Systems and Software* Vol 17 (1992) pp 119–127

85 Gilb, T *Software metrics* Winthrop Publishers (1977)

86 Adamov, R and Richter, L 'A proposal for measuring the structural complexity of programs' *J. Systems and Software* Vol 12 (1990) pp 55–70

87 Oviedo, E I 'Control flow, data flow and program complexity' *Proc. COMPSAC 80* (1980) pp 146–152

88 McCabe, T J and Butler, C W 'Design complexity measurement and testing' *Comm. ACM* Vol 32 No 12 (1989) pp 1415–1425

89 Tsai, W T, Lopez, M A, Rodriguez, V and Volovik, D 'An approach to measuring data structure complexity' in *Proc. COMPSAC '86* (1986) pp 240–246

90 Prather, R E 'An axiomatic theory of software complexity measure' *Computer J.* Vol 27 (1984) pp 340–347

91 Henderson-Sellers, B and Tegarden, D 'Clarification concerning modularization and McCabe's cyclomatic complexity' *Comm. ACM* Vol 37 No 4 (1994) pp 92–94

92 Piwowarski, P 'A nesting level complexity measure' *ACM SIGPLAN Notices* Vol 17 No 9 (1982) pp 44–50

93 Henderson-Sellers, B, Pant, Y R and Verner, J M 'Cyclomatic complexity: theme and variations' *Australian J. of Information Systems* Vol 1 No 1 (1993) pp 24–37

94 Sagri, M 'Rated and operational complexity of program—an extension to McCabe's theory of complexity measure' *ACM SIGPLAN Notices* Vol 24 No 8 (1989) pp 8–12

95 Shepperd, M 'Early life-cycle metrics and software quality models' *Inf. and Soft. Technol.* Vol 32 No 4 (1990) pp 311–316

96 Bieman, J M and Debnath, N C 'An analysis of software structure using a generalized program graph' *Procs. COMPSAC '85* (1985) pp 254–259

97 Tanik, M M 'A comparison of program complexity prediction models' *ACM SIGSOFT, Software Engineering Notes* Vol 5 No 4 (1980) pp 10–16

98 Troy, D A and Zweben, S H 'Measuring the quality of structured designs' *J. Systems and Software* Vol 2 (1981) pp 113–120

99 Basili, V R and Turner, A J 'Iterative enhancement: a practical technique for software development' *IEEE Trans. Soft. Eng.* Vol 1 No 4 (1975) pp 390–396

100 Yau, S S and Collofello, J S 'Some stability measures for software maintenance' *IEEE Trans. Soft. Eng.* Vol 6 No 6 (1980) pp 545–552

101 McClure, C L 'A model for program complexity analysis' *3rd Int. Conf. Soft. Eng.* Atlanta (May 1978) pp 149–157

102 Atwood, M E and Ramsey, H R 'Cognitive structures in the comprehension and memory of computer programs: an investigation of computer program debugging' *Tech. Rep. TR-78-A21* (1978) US Army Res. Ins. for the Behavioral and Social Sciences, Alexandria, VA

103 Li, H F and Cheung, W K 'An empirical study of software metrics' *IEEE Trans. Soft. Eng.* Vol SE-13 No 6 (1987) pp 697–708