Complexity Control in Logic-based Programming

Z. MARKUSZ
The University of Calgary, Department of Computer Science, Calgary, Alberta, Canada T2N 1N4.

A. A. KAPOSI
Polytechnic of the South Bank, Department of Electrical and Electronic Engineering, Borough Road, London, England SE1 0AA.

There is growing awareness of the high risk and excessive cost of poor-quality software design. The problem is especially critical in complex programs where design errors are most frequently committed, and latent errors are particularly difficult to diagnose and correct.[21] The origin of this 'software crisis' lies in the late realisation that there is much more to good software design than knowledge of a programming language. Structured programming and the various software design methodologies seek to control software quality by imposing a discipline on the designer which controls the complexity of design tasks and supplements the rules of the programming language.

This paper outlines the process of evolving a complexity-controlling software design methodology, based on the general principles of sound engineering design, but devised particularly for logic-based programming. It reports on experience with the use of the methodology[16, 27, 30, 5, 31] and compares the complexity properties of programs designed using different methods.

An example is given to show how complexity control may also be achieved retrospectively, or in the course of software maintenance. The procedure is to measure the complexity parameters of a finished program, identifying its most complex parts; these are then reconstructed as hierarchical structures of simple autonomous components, while maintaining functional equivalence.

Based on experience, the paper proposes further refinements of the well-tried complexity measures, suggesting the next stage of evolution of the complexity-controlling methodology for logic-based programming.

Finally, the paper proposes areas for further research into complexity control and its applications to logic-based programming.

1. BACKGROUND TO THE MEASUREMENT AND CONTROL OF SOFTWARE COMPLEXITY

In their interesting paper[3] Curtis *et al.* propose a distinction between two types of software complexity: computational and psychological.

Computational complexity takes account of the quantitative aspects of the algorithm which solves a given problem, estimating the speed of the execution of a program. By contrast, psychological complexity measures the difficulty of the processes of design, comprehension, maintenance and modification of the program itself.

In this paper we concentrate attention on psychological complexity in the above sense. We shall look for objective and quantifiable indicators of the complexity of the design of programs, and regard these as predictors of the cost effectiveness and length of useful life of the software as a product. By devising the complexity measures appropriately, and keeping their values within well-defined bounds, we shall claim to achieve control over important aspects of software quality.

Several attempts have been made in recent years to propose representative and objectively quantifiable measures of psychological complexity. Perhaps the best known of these is Halstead's complexity theory,[11] which measures the effort of program generation and assumes that to be a measure of complexity. He identifies four parameters:

$n1$, the number of different operators within a program;

$n2$, the number of different operands within the program;

$N1$, the total number of occurrences of operators;

$N2$, the total number of occurrences of operands.

All four parameters are directly countable from the program listings and Halstead argues his way to the formula:

$$E = \frac{n1\, N2\, (N1+2) \log2\, (n1+n2)}{2n2}$$

where E is the measure of complexity.

The value and validity of Halstead's theory was questioned by Frewin and Hamer.[9] Curtis' research group[3] undertook its detailed evaluation, based on its application in programming practice. They found that Halstead's measures did not capture all aspects of psychological complexity. In particular, Halstead's measures take no account of structural properties of programs, thus giving no consideration to the complexity-controlling effects of modern methods of software design (e.g. [35, 29] and [6]).

McCabe[29] adopted a completely different approach to software complexity. He developed a theory, based on the modelling of programs as directed graphs; he then measured the complexity of the program by the cyclomatic complexity of their digraph. McCabe's work was refined and developed by a number of authors, including Myers,[32] Oulsnam,[33] Williams[39] and Prather.[34] Since these methods concentrated on structural properties of programs, in principle they provided suitable support for structured software design. However, close examination revealed the arbitrariness of the methods, stemming from lack of rigour in the formulation of the theories.[35, 38, 22, 23] These weaknesses undermine confi-

dence in any complexity measures which may be based upon these theories.

McCabe's graph-theoretic approach inspired the search for a rigorous method of measuring the structural complexity of software.[4, 2] The research culminated in a generalised mathematical theory of structured programming by Fenton, Whitty and Kaposi.[7] The theory provides methods for designing software of controlled structural complexity; it also offers procedures for analysing existing software and for reconstructing it automatically in complexity-controlled form. Whilst it has a broad scope of applicability, the theory has been developed in the context of conventional rather than logic-based programming languages.

The notion of a graph-theoretic approach to complexity control has been adopted by workers in software design and other fields,[8, 36, 17, 1, 13, 18] and research is in progress into general-purpose complexity-controlling systems design methodologies.[20, 24, 12, 37]

2. EVOLUTION OF A COMPLEXITY-CONTROLLING DESIGN METHODOLOGY FOR LOGIC-BASED PROGRAMMING

Knowledge-based systems are finding increasing use in many applications. Such systems may be implemented in PROLOG, a language based on first-order predicate logic, in which the specification of the problem and the means of realising the solution can be expressed. The problem-solving strategy is based on a hierarchical problem-reduction technique, and is implemented by means of deduction and search mechanisms.

Early experience with the use of PROLOG[26] pointed to its potential in a wide range of engineering applications. This experience was gained at a time of growing awareness of the problems of complexity, and led to an attempt to measure the complexity of PROLOG programs.[15]

The rules of the PROLOG language demand the explicit statement of the problem on hand, and the composition of the solution as a strict hierarchical structure of related and explicitly specified parts called 'partitions'. Each partition could be considered as an autonomous entity, hence the complexity of the designer's task could be related to the 'local' complexity of partitions rather than to the 'global' complexity of the PROLOG program as a whole. This led to the conclusion that we needed to measure and control local complexity.[16]

The complexity of a partition appeared to depend on the data relating it to its environment, the number of subtasks within it, the relationships among subtasks, and the data flow through the structure. We therefore decided to measure local complexity as a function with these four arguments as parameters.

As an approximation, we proposed the complexity function as the unweighted sum of the complexity parameters. Thus, we defined the complexity of a partition[16] as follows:

$$l = P1 + P2 + P3 + P4 \qquad (1)$$

where

P1 is the number of new data entities in the positive atom of the partition, i.e. in the problem to be solved by the partition;

P2 is the number of negative atoms in the partition, i.e. the number of subproblems into which the problem divides;

P3 is the measure of the complexity of the relations between between negative atoms; we chose the value of this parameter to be initially 0, adding 2 for each recursive call (a recursive call is considered to have higher complexity than simple AND or OR partitions, because of the potential for infinite recursion);

P4 is the number of new data entities in the negative atoms, i.e. the number of local variables linking the subproblems.

PROLOG syntax, together with the use of the complexity parameters and complexity function, is demonstrated on a program extract, reproduced from [16], using DEC10 PROLOG syntax (Fig. 1).

Program extract

```
task(I, O):- task1(I, O),
             task2(X, O).
task1([Q, S], X):-task3(Q, Y)
             task4([S, Y], X).
```

Complexity parameters	Comments
p1 = 2	I decomposed into Q and S
p2 = 2	*task1* defined in terms of *task3* and *task4*
p3 = 0	*task1* is an AND partition
p4 = 1	New argument Y introduced

Local complexity: $l = p1 + p2 + p3 + p4 = 5$

Figure 1. Example of complexity analysis

Range	Complexity bound	Complexity code of partition
$1 \leqslant l \leqslant 3$	Trivial	T
$4 \leqslant l \leqslant 7$	Simple	S
$8 \leqslant l \leqslant 17$	Complex	C
$18 \leqslant l$	Very complex	V

Figure 2. Complexity bounds

One could then define program complexity as a function with partition complexities as its arguments. We proposed the following definition.

The global complexity of a PROLOG program is the unweighted sum of the local complexity of all n constituent partitions, i.e.

$$g = \sum_{i=1}^{n} li$$

where li is the local complexity of the ith partition.

When used for analysing PROLOG programs, the local complexity measure reflected the intuitive feeling of experienced designers about the relative complexity of design tasks. Global complexity measured program size, but was, as hoped, quite remote from the complexity of any individual local task.

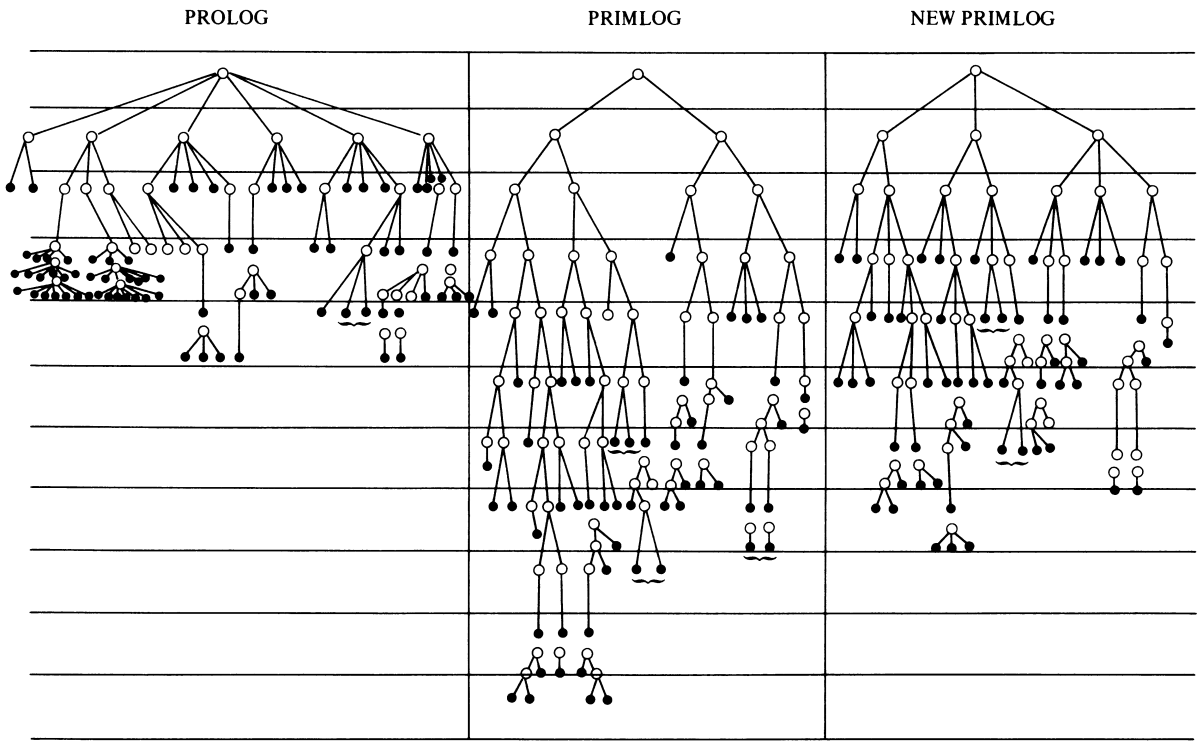
Retrospective analysis of programs also revealed that task complexity was quite out of control. We sorted partitions into four 'complexity bands' according to the value of their complexity function (Fig. 2).

The rationale for the choice of range boundaries is arbitrary but fits in with psychological theory ('magic number 7'). Fig. 5 shows extension to even higher complexity ranges.

In programs which had been designed for real engineering applications the mean value of the complexity function was found to be quite high. Even more worrying

Table 1. Statistics of the three program versions

	Number of partition		
	PROLOG	PRIMLOG	NEW PRIMLOG
Degree of local complexity			
Trivial (T)	4	21	10
Simple (S)	9	25	27
Complex (C)	8	7	6
Very complex (V)	3	—	—
Average measure of local complexity	10 (C)	5 (S)	5.7 (S)
Complexity measure of most complex partition	34 (V)	11 (C)	11 (C)
Total no. of partitions in program	24	53	43
Global complexity	245	266	246
No. of hierarchical levels of the program	4	10	7



Key: ○, new partition
●, partition/clause already accounted for or built in (terminal)

Figure 3. PROLOG, PRIMLOG and NEW PRIMLOG equivalents of a CAD program for architectural application.

was the high deviation from the mean, with one or two partitions being excessively complex, usually those dealing with the most problematical areas of the design. This was considered an indicator of error-prone program features.

Our confidence in the value of complexity control was boosted by its aiding the diagnosis of latent logical errors in high-complexity partitions of programs already installed in the field which had been carefully designed and tested by the experts, but not with the aid of complexity control. Clearly, the rules of the PROLOG language provided too liberal a regime, and needed to be supplemented by a stricter discipline of coding rules, based on a deep understanding of the causes of complexity, its measurement in the course of design practice and a method for its control.

Control could be achieved by setting upper bounds on the value of each of the complexity parameters and upon the complexity function itself, but the question was how to choose the value of these bounds. Rather than relying on common sense alone, we decided on evolving design rules by a deliberate strategy, as follows.

A 'minimal complexity' method called PRIMLOG (PRIMitive proLOG),[15] was defined, constraining the

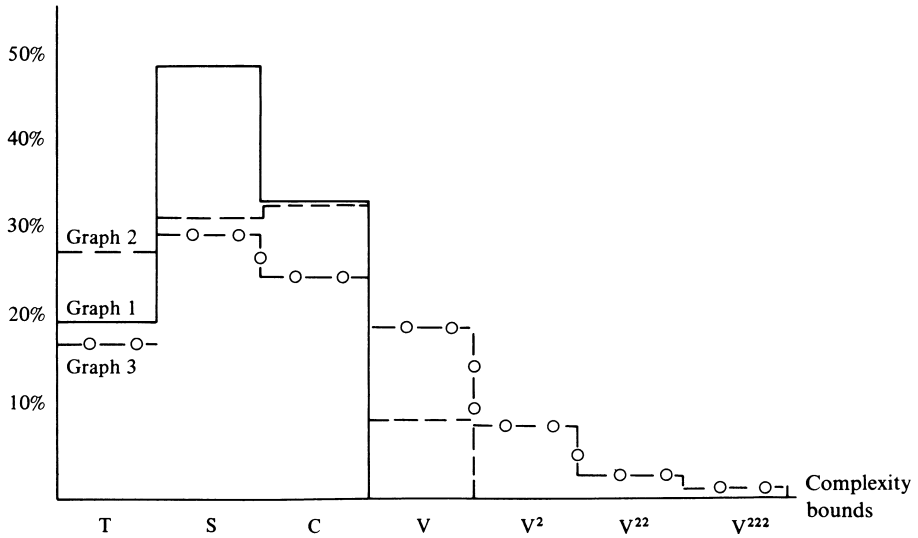


Figure 4. Complexity distribution of three programs. Graph 1, building design program; graph 2, tower design program; graph 3, fixture design program.

designer to the strict essentials consistent with hierarchical refinement (see Appendix A). PRIMLOG was considered a first approximation to a usable design methodology.

PRIMLOG was imposed as a discipline for designing CAD programs, and programmers' reactions were noted. In view of these, the rules were relaxed in those respects where they raised the most justified complaints, while still retaining complexity control. This resulted in the proposition of NEW PRIMLOG, as Mark II in the evolution of the methodology. The design rules of NEW PRIMLOG were formulated simply as restrictions on normal PROLOG syntax, constraining each of the four parameters of the complexity function, as shown in Appendix B. These rules were further supplemented by the strong recommendation that designers should avoid the use of 'very complex' partitions. This effectively constrained the value of the global complexity function itself.

Controlled experiments were conducted to compare the unconstrained PROLOG design with that obtained by PRIMLOG and NEW PRIMLOG, all three program versions meeting the same specifications. The objects of these experiments were, by necessity, programs of modest scale, but they were nevertheless useful CAD programs in their own right. The outcome of one of these experiments was reported in [16]. The effect of complexity control upon the tree of hierarchical refinement is shown in Fig. 3, and some of the statistics of the three program versions are given in Table 1. The results are discussed in some detail in [16].

As an outcome of these trials, sufficient confidence had been established to introduce the rules of NEW PRIMLOG into some areas of engineering practice, enlisting the help of some designers to record their experiences and opinions.

Further experience by users revealed the value of NEW PRIMLOG as an aid to design, maintenance and management of software development. It also showed the way to simplify the methodology without the loss of complexity control; instead of individually controlling each of the four arguments of the complexity function of Equation 1, as the NEW PRIMLOG rules sought to do, it was found sufficient only to keep the value of the function itself within bounds. Thus, the Mark III version of the methodology was established, and was put to use.[28] This paper summarises the insight gained in more than two years and in the course of several projects (e.g. [27, 30, 5, 31] and [10]). Recent experience showed a deficiency of the complexity function of Equation 1 in capturing all of the aspects of task complexity. Section 5 of the paper discusses recommendations for further refinement, proposed by user-designers. This will be evaluated by the research team, taking into account conclusions drawn from systems research. The procedure of successive refinement will continue as the methodology is evaluated in practice and is developed by systems research.

3. EXPERIENCE WITH COMPLEXITY CONTROL. TECHNICAL CONSIDERATIONS

User experience is discussed with reference to three of the CAD programs designed by logic-based programming, as follows.

A 'building design' program, concerning the modular design of many-storeyed dwelling houses built of prefabricated elements ([27], column 1 of Table 2, graph 1 of Fig. 4). The program was designed in a strictly top-down manner following the rules of NEW PRIMLOG.

A 'tower design' program, concerning the design of building blocks for supporting mechanical engineering parts in the course of machining,[5] column 2 of Table 2, and graph 2 of Fig. 4). The program was designed by using a mixed top-down and bottom-up strategy and by use of the Mark III version of complexity control.

A 'fixture design' 5 program, related to the same application area (column 3 of Table 2 and graph 3 of Fig. 4). The program was designed by an expert programmer, interested in, but not subjecting to, complexity control. Mixed top-down/bottom-up strategy was used.

Strict top-down design under NEW PRIMLOG tends to lead to low average local complexity. This has benefits of short development time and ease of diagnosing and correcting errors. However, if local complexity drops to very low values then the number of hierarchical levels, and the number of intermediate subproblems, tends to

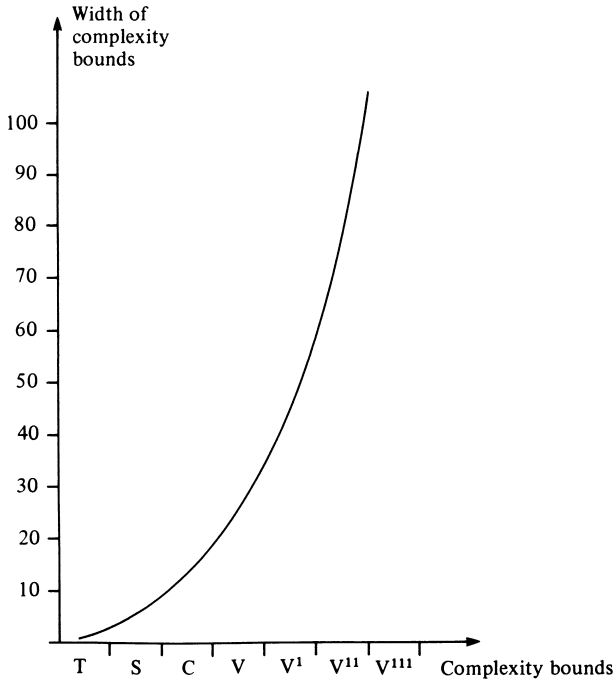


Figure 5. Extension of complexity bounds.

increase, and there is a corresponding increase in the number of logical errors. There seems to be some optimal value at which average local complexity should be kept so as to optimise program quality. Clearly, it seems advantageous not only to keep average complexity near such an optimum but also to avoid undue deviation from the average within a given program. There will be reference to this later in this paper.

When the designer used a mixed top-down and bottom-up approach, and only controlled local complexity rather than its individual parameters, the tendency was for average local complexity to increase. It is interesting to compare the statistics of the 'building design' and 'tower design' programs (columns 1 and 2 of Table 2). The latter is a smaller program, as indicated by the lower global complexity, but its average local complexity is higher and the number of partitions is very much smaller. Errors were fewer but they were harder to find; both development time and testing time were higher than was the case for the larger, simpler structure of the 'building design' program.

Design followed a mixed strategy without complexity control. The size of this program is virtually the same as that of 'building design'. Full information is not available, but Table 2 and Fig. 4 show the startling difference in programming style.

Analysis of programs which had been designed without complexity control showed the need to subdivide the 'very complex' band of the value of complexity functions. By plotting the width of bands on a linear scale, the graph of Fig. 5 was obtained. The 'very complex' region is far too wide and we therefore created 4 subdivisions: V, V′, V″, V‴. The most complex partition of the 'fixture building' program (1 = 156) falls in the last of these bands.

It is interesting to note the uniform programming style resulting from use of NEW PRIMLOG and top-down design. In the 'building design' program 71% of partitions are 'Simple' or 'Complex', only 19% are 'Trivial' and none is 'Very complex'.

This is considered an attractive feature and is likely to lead to good performance in the field, provided that the average complexity is set at a more appropriate level. 'Too many' 'trivial' partitions may obfuscate the true

Table 2. Complexity analysis of three CAD programs

	Building design (1)	Tower design (2)	Fixture design (3)
Number of partitions	251	164	108
Global complexity	1636	1349	1629
Average local complexity	6.5	8.2	15
Trivial partitions T (1–3)	49 (19%)	43 (27%)	18 (17%)
Simple partitions S (4–7)	120 (48%)	3 (32%)	32 (29%)
Complex partitions C (8–17)	82 (33%)	55 (33%)	27 (25%)
Very complex partitions V (17–34)	— —	13 (8%)	21 (19%)
Very complex partitions VI (35–59)	— —	— —	7 (7%)
Very complex partitions VII (60–100)	— —	— —	2 (2%)
Very complex partitions VIII (101 – any)	— —	— —	1 (1%)
Maximal complexity	17	23	156
Number of hierarchical levels	13	8	8
Method of developing the programs	NEW PRIMLOG	With complexity control	Without complexity control
Program development (man-day)	24	28	—
Program testing	49	52	—
Total time	73	80	—
Number of semantic errors	42	25	—

Table 3. Complexity analysis of a reconstructed partition

No.	Partition name	Type	P1	P2	P3	P4	l	Complexity code
1	Task	CASE	4	28	—	23	55	V[1]
1	Task	CASE	3	3	—	—	6	S
2	Task 1	AND	2	6	—	5	13	C
3	Task 2	AND	2	5	—	2	9	C
4	Task 3	AND	2	4	—	3	9	C
5	Value	AND	—	4	—	3	7	S
6	Task33	AND	—	2	—	3	5	S
7	Data	AND	—	3	—	—	3	T
8	Count	AND	—	5	—	4	9	C

Global complexity of reconstructed TASK: 61

$$\tilde{l} = \frac{61}{8} = 7.6 \quad \text{Simple}$$

function of a program section as well as impose an execution efficiency penalty. Too many 'very complex' partitions are also hard for the programmer to understand and may increase the opportunity for error. Our experience shows that complexity levels between 7 and 10 may be considered 'appropriate' (to expert PROLOG programmers).

4. SOFTWARE MAINTENANCE/RECONSTRUCTION UNDER COMPLEXITY CONTROL

We have selected one of the very complex (V′ band) partitions of the 'fixture building' program, which bears the name of TASK, to illustrate hierarchical reconstruction in detail. In the original design the local complexity of the partition was 55. The reconstruction produced a structure of 8 partitions, with an average complexity of 7.6 (nearly 'Simple'), and a global complexity of the new TASK of 61, an increase of just over 10% (see Table 3).

Among the advantages of the reconstruction was that it became possible to isolate a partition named VALUE which was found to have repeated applications in TASK and in two other places elsewhere in the program. This partition was defined as a separate entity and its use contributed to the elegant design of both TASK and other parts of the program. Simplifications also occurred in the hierarchical reconstruction of the third argument of the partition. To illustrate this, and help in tracing the two versions of the design of the partition, the relevant parts of the program texts are enclosed in Appendix C.

The number of partitions in the reconstructed partition increased by 7 and the number of hierarchical levels by 2. There are some indications that in some cases the reconstructed programs may carry higher run-time overheads than the original (see also [16]). It seems that such overheads may correlate with global (rather than local) complexity and with the number of partitions in the program. If future experience shows that this is so, there would be an even stronger case for encouraging a more uniform design style by setting both upper and lower bounds in emphasising the importance of run-time efficiency when evaluating software quality. In the case of off-line applications such as CAD, cost-effectiveness of the life-cycle performance is far more likely to correlate with low task complexity than with run-time efficiency.

5. THE NEED TO REFINE THE COMPLEXITY-CONTROLLING METHODOLOGY

It was mentioned earlier that when comparing the values of the complexity function with the designers' relative difficulties in developing program partitions, anomalies were found. This is because the complexity measures of Equation (1) fail to penalise the complexity of relations between new data entities, hence designers are not deterred from using various and highly complex compound operators within the arguments of positive atoms. A case in point is the third argument of TASK (see Appendix C).

On the basis of this observation, designers proposed the introduction of new parameters into the complexity function of equation (1), accounting for the variety and complexity of operators on the data. This proposition is reasonable: it produces an even-handed treatment of subproblems and data, whereas the earlier version of the complexity function does not take data relations into account. Accordingly, the proposed complexity function would become:

$$l = P1 + P2 + P3 + P4 + f1 + f2 \qquad (2)$$

$P1+P2+P3+P4$: the number of new subproblems and the relations between them

$f1+f2$: the number of new data entities and the relations between them

with

$f1$ as the number of different operators (infix or prefix) within the partition, and

$f2$ as the measure of the complexity of the relationships among the new data entities.

A more adequate notation would be:

$$l = t1 + t2 + d1 + d2 + d3 + d4 \qquad (3)$$

with parameters of corresponding interpretation

't' standing for TASK, and

'd' standing for DATA ENTITY.

Table 4. Examples of two new complexity parameters

	f1	f2	f1+f2	Example
Simple prefix operator or function notation	1	0	1	f1(L, D, H)
Binary infix operator	1	0	1	i . P
List denoted by • infix operator (tree structure used in recursion)	1	2	3	X1 . X2 . X3 . nil
Two different nested operators	2	3	5	f3(L, P1 . P2, H)
Three different nested operators* etc.	5	5	10	t(N, f2(Y, N . i, H), zax (N, Y, s(X, Y, Z))

All examples refer to parts of program under analysis in this paper in Appendix C.
* The structure includes five different operators: t, f2, ., zax and s. Of these only three are nested.

Table 5. Complexity analysis as in Table 3, considering two additional parameters

No.	Partition name	Type	t1	t2	d1	d2	d3	d4	l	Complexity degree
1	Task	CASE	28	—	4	23	4	9	69	V″
1	Task	CASE	3	—	3	—	3	—	9	C
2	Task1	AND	6	—	2	5	1	—	14	C
3	Task2	AND	5	—	2	2	1	—	10	C
4	Task3	AND	4	—	2	3	1	—	10	C
5	Value	AND	4	—	—	3	—	—	7	S
6	Task33	AND	2	—	—	3	2	3	10	C
7	Data	AND	3	—	—	—	—	—	3	T
8	Count	AND	5	—	—	4	—	—	9	C

Global complexity of reconstructed TASK: g = 72.

$$\frac{72}{8} = 9 \quad \tilde{l} = 9.$$

Table 6. Comparing original and improved functions applied to three CAD programs

Name of the program	Number of partitions	Old value g of global complexity	$\tilde{l}$	New value g′ of global complexity	g′-g	$\tilde{l}'$
Building design	251	1636	6.5	2010	374	8
Tower design	164	1349	8,2	1674	325	10, 2
Fixture design	108	1629	15	1834	205	17

We may now introduce a scale of 'charges' upon the new complexity parameters, such as is shown in Table 4.

As an experiment with the newly proposed complexity function, we may now re-compute the local complexity of TASK in both its original and reconstructed version. The results are shown in Table 5. It is interesting to note that the global complexity of the reconstructed partition only differs from the complexity of the original by (72–68) = 4, although seven new part-problems were included. This is considered an encouraging indicator of the effectiveness of complexity control.

To check the performance of the newly proposed complexity measure against the intuitive evaluation of complexity by designers, we have re-computed the complexity statistics of all three programs of Table 2. The new measures are summarised in Table 6. The conclusions are that the new measures represent a considerable improvement in quantifying the difficulty of design tasks.

6. CONCLUSIONS AND FURTHER WORK

The complexity measures proposed here may be applied in two ways:

to prevent errors, controlling the quality of newly designed software; as a means of quality assurance, to detect the areas of potential design weakness in existing

software, and to guide the process of reconstruction into functionally equivalent but complexity-controlled form.

Although the measures are still rudimentary and need further refinement, experience shows them to be effective in both these contexts.

Further work is needed to continue the evaluation and refinement of the measures, in the light of experience with their use; to investigate methods of reconstruction under conditions of stronger equivalence, making use of the results reported in Ref. 7.

REFERENCES

1. R. C. Chandra, Design methodology for microprocessor-based process control systems, Ph.D. Thesis, Polytechnic of the South Bank (1980).
2. D. F. Cowell, D. F. Gillies and A. A. Kaposi, Synthesis and structural analysis of abstract programs, *The Computer Journal* **23** (3), 243–247 (1981).
3. B. Curtis, S. B. Sheppard, F. Milliam, M. A. Borst and T. Love, Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics, *IEEE Transactions on Software Engineering* **SE-5** (2) (1979).
4. C. H. Dowker and A. A. Kaposi, The geometry of flow-charts, Internal Report C2-1979, Polytechnic of the South Bank (1979).
5. J. Farkas, J. Fileman, A. Markus and Z. Markusz, 'Fixture design by PROLOG' **MICAD-82**, Paris (1982).
6. M. B. Feldman, Data abstraction, structured programming, and the practicing programmer, *Software – Practice and Experience* **11**, 679–710.
7. N. E. Fenton, W. R. Whitty and A. A. Kaposi, A generalised mathematical theory of structured programming, Internal Report RW-14, Polytechnic of the South Bank, Department of Electrical and Electronic Engineering (1983).
8. Franksten, Falster and Evans, Qualitative aspects of large-scale systems. *In Lecture Notes in Control and Information Science*. Springer Verlag, Heidelberg (1979).
9. G. Frewin and D. Hamer, M. H. Halstead's software science – a critical examination, ITT Technical Report No. STL 1341 (1981).
10. E. Halmay and P. Gero, PROGART, a computerised assistant for the programming instructor, Research Report, CSO International Computer Education and Information Centre, Budapest, Hungary (1981).
11. M. H. Halstead, *Elements of Software Science*. Elsevier, New York (1977).
12. R. N. Holden and A. A. Kaposi, A graphical language for modelling multilevel hierarchical systems, *IEIP Workshop*, La Baule, France (1980).
13. R. N. Holden, A set of complexity measures for PASCAL, International Report R-8, Polytechnic of the South Bank, Department of Electrical and Electronic Engineering (1982).
14. A. A. Kaposi, D. F. Gillies and D. F. Cowell, *The Computer Journal* **22** (2), 1980 (Letter to the Editor) (1979).
15. A. A. Kaposi, L. Kassovitz and Z. Markusz, PRIMLOG, a case for augmented PROLOG programming *Proc. Informatica*, Bled, Yugoslavia (1979).
16. A. A. Kaposi and Z. Markusz, Introduction of a complexity measure for control of design errors in logic-based CAD programs, *Proceedings, CAD* **80**, Brighton (1980).
17. A. A. Kaposi and R. C. Chandra, A methodology for the description of real-time digital systems, *IFIP Invitation Workshop*, La Baule, France (1980).
18. A. A. Kaposi and R. C. Chandra, A methodology for the description of real-time digital systems, *IFIP Workshop*, 885–889, La Baule, France (1980).
19. A. A. Kaposi and R. C. Chandra, A systems approach to the analysis and design of logical controllers, *Proceedings of the International Conference on Systems Engineering* **1**, Coventry (1980).
20. A. A. Kaposi and G. Rzevski, On the aims and scope of a systems design methodology, *Proceedings of the 5th European Meeting on Cybernetics and Systems Research*, 123–127, Vienna (1980).
21. A. A. Kaposi and R. N. Holden, An engineer's view on the assurance of software quality, *Electronics & Power*, 508–510 (1982).
22. A. A. Kaposi and W. R. Whitty, Internal Report E7-20887, Polytechnic of the South Bank, Department of Electrical and Electronic Engineering (1982).
23. A. A. Kaposi, Internal Report E6-15982, Polytechnic of the South Bank, Department. of Electrical and Electronic Engineering (1982).
24. A. A. Kaposi, 'Complexity measures for designing engineering systems'. Paper for *Systems Engineering*, II, Coventry (1982).
25. J. F. Leathrum, "A Design Medium for Software" *Software – Practice and Experience* **12**, 497–503 (1982).
26. Z. Markusz, How to design variants of flats using the programming language PROLOG, *Proceedings, IFIP*, Toronto, Canada (1977).
27. Z. Markusz, Design in logic, *Computer Aided Design* **14** (6) 335–343 (1982).
28. Z. Markusz and A. A. Kaposi, A design methodology in PROLOG programming, *Proceedings, First International Logic Programming Conference*, 139–145, Marseille, France (1982).
29. T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* **SE-2** (4), 308–320 (1976).
30. B. E. Molnar and A. Markus, Logic programming for the modelling of machine parts, *Compcontrol '81*, Varna, Bulgaria (1981).
31. B. E. Molnar and A. Markus, Logic programming in the design of production control systems, *Compcontrol '81*, Varna, Bulgaria (1981).
32. G. J. Myers, An extension to the cyclomatic measure of program complexity, *Sigplan Notices* (1977).
33. G. Oulsnam, Cyclomatic numbers do not measure complexity of unstructured programs, *Information Processing Letters* **9** (5) (1979).
34. R. E. Prather and S. G. Giulieri, Decomposition of flow-chart schemata, *The Computer Journal* **24** (3) 258–262 (1981).
35. S. R. Schach, A unified theory for software production, *Software – Practice and Experience* **12**, 683–689 (1982).
36. X. Schizas, A graph-theoretic approach to the analysis of large-scale systems, Ph.D. Thesis, London University (1981).
37. E. Secco, Design methods for embedded computer systems, Internal Report Z-4, Polytechnic of the South Bank, Department. of Electrical and Electronic Engineering.
38. A. Shafibegly-Gray and W. R. Whitty, Comment on 'Decomposition of flowchart schemata', *The Computer Journal* **25** (4), 495, letter to the Editor (1982).
39. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form, *The Computer Journal* **21** (2), 161–167 (1978).

APPENDIX A

PRIMLOG RULES

TASK:	:- R (d1, d2), P (r1, r2).
AND:	R (t1, t2):- A (d1, d2), B (r1, r2).
OR:	R (t1, t2):- A (x1, x2). R (d1, d2):- B (r1, r2).
RECURSION:	R (t1, t2). R (d1, d1):- A (x1, x2), R (r1, r2).
DATABASE:	A (t1, t2). A (r1, r2). ⋮ A (d1, d2).
where	R, A, B, P are relation symbols or atom names,
and	t1, t2, x1, x2, d1, d2, r1, r2 are terms.

Note: To minimise complexity at every level, compound structures are permitted to contain at most two substructures. This results in a binary decomposition of the problem.

APPENDIX B

NEW PRIMLOG RULES

(a) Extension of AND partition
A:- B1, B2, B3.
if one of the negative atoms B1, B2, B3 is a built-in procedure or matches with a DATA BASE assertion.

(b) Extension of OR partition
A:- B1, B2.
A:- C1, C2.
if one of the negative atoms B1, B2, C1, C2 is a built-in procedure or matches with a DATA BASE assertion.

(c) Introduction of CASE partition
A:- B1.
A:- B2.
⋮
A:- BN.
where N is arbitrary.

APPENDIX C

Original and reconstructed program fragments

```
/☆ partition 'task' is called in this way: ☆/

task(X, Y, T, N, M, C).

/☆ partition 'task' – original version ☆/

task(X, Y,  f1(L, i.P, H), X, M, C):-
  data3(P, A, B),
  plus(Y, A, E),
  round(E, 50, G, D),
  less(D, 10),
  plus(G, B, M),
  data0(i, Q),
  data0(P, R),
  times(Q, R, F),
  times(45, F, C).

task(X, Y,  f2(1, P.i, H),  N, M, C):-
  round(Y, 50, M, D),
  less(D, 12),
  data2(P, D2),
  plus(X, D2, N),
  data0(P, Q),
  data0(i, R),
  times(Q, R, F),
  times(45, F, C).
task(X, Y,  f3(L, P1.P2, H), N, M, C):-
  data1(P2, D2),
  plus(Y, D2, T),
task(X, T, f2(L, P1.i, H), N, M, CC),
  data0(i, F1),
  data0(P1, F2),
  data0(P2, F3),
  times(F2, F3, Q),
  times(F1, F2, P),
  times(18, Q, R),
  times(54, P, S),
  plus(R, S, C).

/☆ partition 'task' – rewritten version ☆/

task(X, Y,  f1(L, D, H),  N, M, C):-
  task1(X, Y,  f1(L, D, H),  N, M, C).

task(X, Y,  f2(L, D, M),  N, M, C):-
  task2(X, Y,  f2(L, D, M),  N, M, C).

task(X, Y,  f3(L, D, M),  N, M, C):-
  task3(X, Y,  f3(L, D, M),  N, M, C).

task1(X, Y,  f1(L, i.P, H),  X, M, C):-
  data3(P, A, B),
  plus(Y, A, E),
  round(E, 50, G, D),
  less(D, 10),
  plus(G, B, M),
  value (P, C).

task2(X, Y,  f2(L, P.i, H),  N, M, C):-
  round(Y, 50, M, D),
  less (D, 12),
  data2(P, D2),
  plus(X, D2, N),
  value(P, c).

task3(X, Y,  f3(L, P1.P2, H),  N, M, C):-
  data(P2, D2),
  plus(Y, D2, T),
  task2(X, T,  f2(L, P1.i, H),  N, M, CC),
  task33(P1, P2, C).

value(P, c):-
  data0(i, Q),
  data0(P, R),
  times(Q, R, F),
  times(45, F, C).

task33(P1, P2, C):-
  data(P1, P2, F1, F2, F3),
  count(F1, F2, F3, C).

data(P1, P2, F1, F2, F3):-
  data0(i, F1),
  data0(P1, F2),
  data0(P2, F3).

count(F1, F2, F3, C):-
  times(F2, F3, Q),
  times(F1, F2, P),
  times(18, Q, R),
  times(54, P, S),
  plus(R, S, C).
```